C语言基础

C语言是在 70 年代初问世的。1978 年由美国电话电报公司(AT&T)贝尔实验室正式发表了C语言。早期的C 语言主要是用于Unix 系统。由于C语言的强大功能和各方面的优点逐渐为人们认识，到了八十年代，C 开始进入其它操作系统，并很快在各类大、中、小和微型计算机上得到了广泛的使用，成为当代最优秀的程序设计语言之一。由于 C 语言与 Unix 的密切关系，C 语言也成为 Linux 下的最流行的高级语言，实际上 Linux 系统的绝大部分代码是用C语言写的。本章将对C语言的用法做一简单介绍。

0.1 C 语言特点

C语言的强大生命力归功于C语言的优秀特点，归纳起来，C语言主要特点如下：

1）C语言非常简洁，而且重视实用性，关键字少，一共只有 32 个。因此 c 语言书写的程序长度短，减少了输入的工作量。

2）C 语言表达能力很强，其他高级语言难于表达的运算表达式使用 C 语言可以很容易地实现。这是内于 c 语言的运算符非常丰富，共有 34 种运算符，包含的范围很广泛，可以用来构建类型多样化的表达式。在C语言中括号、赋值符号、强制类型转换等被视为运算符，灵活使用各种运算符和表达式可以实现非常复杂的运算功能。

3）C语言易于描述复杂的数据结构。C语言的数据类型有；整型、实型、字符型、数组类型、指针类型、结构体类型、共用体类型等，可以方便的描述各种常用的数据结构，如链表、树、图等，这使得C语言在开发大型的系统和应用程序方面有很强的优势。C语言的一大特色就是可以进行灵活的指针操作，这就使C语言程序可以完成其他高级语言难于实现的功能，并保证了极高的运行效率。

4）C语言非常接近于硬件，允许直接访问物理地址，并能进行位操作，能够以更容

易理解的方式实现汇编语言的大部分功能。C 语言具有高级语言和低级语言的双重特性，既可以作为系统的描述和开发语言，又是通用的程序设计语言。现在流行的操作系统一般由 C 语言编写绝大多数的代码，而非常接近硬件并对性能要求严格的部分才由汇编来完成。

5）C 语言编写的代码编译生成的目标代码质量非常高，程序运行效率一般只比汇编程序生成的目标代码低 10%—20%。C 语言有非常优秀的编译系统可以选择，程序员可以把大部分的优化工作交给编译程序完成。

6）C 语言具有由函数集合所构成的模块化结构。函数是 C 语言代码的基本构成部分。开发者可以将一个大型程序分割成若干部分或函数，并分别由不同的人员同时编写，因此，C 语言也是一种具有高度结构化和模块化特性的语言。

7）C 语言编写的程序非常容易移植，到目前为止，几乎所有的操作系统平台上都有 C 语言的编译系统，而且 C 语言具有统一的标准，大大简化了软件在不同平台的移植工作。

但是，C 语言也存在缺点，比如：

1）C 语言的语法检查不严格，虽然这样可以给程序员更大的灵活性，但是这样会使程序员养成不良的编程习惯，并导致程序存在隐含的错误。这个特点使得 C 语言对程序员的程序设计思想和技能要求较高。

2）C 语言的指针操作非常灵活，不正确的使用会造成程序运行的严重错误，甚至会造成系统不稳定。

随着 Linux 的不断发展，C 语言作为 Linux 系统的主要编程语言应用越来越广泛，下面将逐步介绍在 Linux 系统上如何运用 C 语言进行软件开发。

0.2 C 语言程序结构

在第 2 章中，已经给出了简单的 C 语言程序例子。在此我们仍以几个简单的例子来

说明 C 语言源程序结构的特点，这几个程序由简到难，表现了 C 语言源程序在组成结构上的特点。从这些例子中了解到组成一个 C 源程序的基本部分和书写格式。

例 0.1：在屏幕上输出"Hello，world！"

```
#include   <stdio.h>
main()   /*主函数*/
{
        printf("Hello world!\n");   /*调用标准库函数，显示引号中的内容*/
 }
```

这是第 2 章中给出的最简单的 C 程序，其执行结果是在终端屏幕上显示一行信息：

```
Hello world!
```

下面，对上面的程序进行说明。

程序的第一行#include 称为文件包含命令，它指的是一个程序把另一个指定文件的内容包含进来，扩展名为.h 的文件称为头文件或首部文件。书写时，可以使用引号也可以用尖括号。例如：

```
#include   "stdio.h"
```

或

```
#include   <stdio.h>
```

都是在程序中把文件 stido.h 的内容(引号或尖括号是一定要的)包含进来。文件名是用双引号还是尖括号，其含义并不一样。使用尖括号时，C 编译系统首先在系统指定的目录中寻找包含文件，如果没有找到，就到当前工作目录中去寻找，这是引用系统提供的包含文件所采用的方法。而使用双引号时，C 编译系统只在用户指定的目录下和当前目录下寻找包含文件。

在程序设计中，文件包含语句是非常有用的。一般 C 系统中带有大量的.h 文件，用户可根据不同的需要将相应的.h 文件包含进来。

stdio.h 是关于标准输入输出的头文件，它是由系统提供的，其中定义了标准输入和输出库函数的许多信息。可以说，一般的 C 程序都离不开这条语句。

所有的 C 程序都是由一个或多个函数构成，其中必须只能有一个主函数 main。程序从主函数开始执行，当执行到调用函数的语句时，程序将控制转移到调用函数中执行，执行结束后，再返回主函数中继续运行，直至程序执行结束。C 程序的函数是由编译系统提供的标准函数（如 printf、scanf 等）和由用户自己定义的函数等。虽然从技术上讲，

主函数不是 C 语言的一个成分，但它仍被看做是其中的一部分。

花括号{}括起来构造函数的语句，称为函数体。在这个例子中，函数体只有一条函数调用语句 printf。和其它高级语言一样，C 语言的语句也是用来向计算机系统发出操作指令的。一条语句经过编译后生成若干条机器指令。一个为实现特定目的的程序应包含若干条语句，即一个 C 程序可以由若干个源程序文件（分别编译的文件模块）组成，一个源文件可以由若干个函数和预编译命令组成，一个函数又由数据定义和执行语句两部分组成。

printf 函数是一个由系统定义的标准函数，可在程序中直接调用。printf 函数的功能是把要输出的内容（本例中是 Hello world!）送到显示器去显示。\n 为转义字符，代表换行的意思，后面还会进一步介绍。

C 语言中一个语句一般占一行，语句的最后跟着一个分号“;”，C 语言规定在每条语句最后必须加一个分号“;”表示该语句的结束。分号称为终结符，单独一个分号也可构成一个语句，这就是空语句。如果不加分号，编译程序会提示错误。

程序中/ * . . . . . * /表示对程序的说明（称为注释），可以方便程序的阅读。注释不参与程序的运行，它可以加在程序的任何位置，注释文字可以是任意字符，如汉字、拼音、英文等。

下面再看一个较为复杂的程序。

例 0.2：输入圆柱体的半径和高度，计算圆柱体的体积。

```
#include   <stdio.h>
main()
{
 float   radius,height,vol;     /*定义浮点型变量*/
 float volume(float,float);     /*声明 volume 函数*/
 printf("Please input the radius and height:\n");
 scanf("%f %f",&radius,&height);         /*调用库函数，从键盘输入 radius,height 的值*/
 vol=volume(radius,height);          /*  调用 volume 函数，计算圆柱体的体积*/
 printf("volume= %f\n",vol);
}
```

```
float volume(float r, float h)   /*定义 volume 函数*/
{
 float   temp;            /*定义函数内部使用的变量 temp*/
 float pi=0.1415926;   /*定义圆周率 pi */
 temp=pi*r*r*h;         /*计算体积*/
 return temp;            /*返回计算出的体积值*/
}
```

例 0.2 比例 0.1 复杂一些，它包含了 main 函数和一个根据半径和高度计算圆柱体体积的 volume 函数。

main 函数的第一条语句是变量说明语句。前面已经说过函数由数据定义和执行语句两部分组成，变量说明即为数据定义的一部分。在 C 语言中，所有变量都必须先说明后使用，说明通常放在函数开始处的可执行语句之前。说明用于声明变量的性质，它由一个类型名与若干所要说明的变量组成，例如：

```
float   radius,height,vol;
```

就定义了 radius，height，vol 是浮点数变量（即我们通常所说的实数变量），其中，float 是类型标识符，表示所列变量为浮点数变量，radius，height，vol 是变量名，在本程序中分别代表圆柱体的半径，高度和体积。关于类型标识符与变量在后面的内容将会详细介绍。

在变量定义语句之后，volume 函数的声明语句。因为函数 volume 是在 main 函数定义之后才定义的，并且在 main 函数中要调用 volume 函数。如果对 volume 函数不加以声明，编译系统会不知道 volume 函数的类型，因而无法编译。函数声明的目的就是告诉编译系统，main 函数里调用的 vol 函数的参数类型以及返回类型，使得程序可以正常编译。函数声明语句中的的第 1 个 float 表示函数的返回值类型是浮点数类型，volume 是函数名，括号中的 2 个 float 说明 volume 函数的参数类型是浮点型。

在 volume 函数声明语句之后是 printf 函数和 scanf 函数，它们用来输出相关信息和获得从键盘输入的圆柱体的半径和高度值，并把值赋给 radius 和 height 变量。

获得圆柱体的半径和高度值之后，main 函数以此作为参数，调用 volume 函数，计算得出圆柱体的体积，并把结果通过赋值运算符“=”赋给变量 vol。

main 函数的最后一条语句是输出圆柱体的体积值。

main 函数后面是函数 volume 的定义，包括函数返回值和函数参数的定义，在本程序中函数返回值类型是 float 型，2 个参数分别代表圆柱体的半径和高度，它们也是 float 型。

volume 函数的第 1 条语句是变量定义语句，定义了一个浮点型变量 temp，用来保存计算得出的结果值。第 2 条语句也是变量定义语句，定义了圆周率 pi 的大小，并且给 pi 赋了初值。第 3 条语句是根据半径和高度，计算圆柱体的体积，并把计算出的值赋给变量 temp，其中“*”代表乘法运算。volume 函数的最后一条语句是返回圆柱体的体积。关于函数的定义和调用在后面会进一步介绍，读者在此只须了解函数可以完成一定的功能，由一系列语句组成即可。

编译链接后，运行本程序时，首先在终端上给出提示：输入半径和高度，从键盘键入数据，如 1，3，按下回车键，程序给出运行结果：

```
Please input the radius and height:
1 3
volume= 9.424778
```

从以上 2 个例子中可以看出，所有的 C 程序为函数模块结构，所有的 C 程序都是由一个或多个函数构成。从技术上讲，纯粹由程序员自己编写的语句构成 C 语言程序是可能的，但这却是罕见的。因为所有的 C 编译程序都提供能完成各种常用任务的函数—函数库（如 printf、scanf 等）。C 编译程序的实现者已经编写了大部分常见的通用函数。当我们调用一个别人编写的函数时编译程序“记忆”它的名字。随后，“链接程序”把我们编写的程序同标准函数库中找到的目标码结合起来，这个过程称为“链接”。

编写程序时用到的函数，许多都可以在标准函数库中找到。它们是可以简单地组合起来的程序构件。编写了一个经常要用的函数之后，可将其放入库中备用。

0.3 数据类型、运算符与表达式

计算机的基本功能是进行数据处理，一种语言支持的数据类型越丰富，它的应用范围就越广。C 语言可以提供丰富的数据类型，不仅能表达并处理基本的数据（如整形、

实数、字符等），还可以组织成复杂的数据结构（如链表、树等）。

在C语言中，数据类型可分为：基本数据类型，构造数据类型，指针类型三大类。基本数据类型可分为整形、浮点型、双精度浮点型、字符型。构造类型可分为数组类型、结构体类型和共用体类型以及枚举类型。在C语言中，每一种数据类型都有一个标识符与之相对应，称之为类型名。

C语言的运算符非常丰富，主要有以下几类：算术运算符、关系运算符、逻辑运算符、位运算符和其他一些用于完成特殊任务的运算符。C语言中运算符和表达式数量之多， 在高级语言中是少见的。正是丰富的运算符和表达式使C语言功能十分完善。 这也是C语言的主要特点之一。

C语言的运算符不仅具有不同的优先级， 而且还有一个特点，就是它的结合性。在表达式中，各运算量参与运算的先后顺序不仅要遵守运算符优先级别的规定，还要受运算符结合性的制约，以便确定是自左向右进行运算还是自右向左进行运算。这种结合性是其它高级语言的运算符所没有的，因此也增加了C语言的复杂性。

0.0.1 标识符

所谓标识符，实际上就是一个字符序列。在C语言中，标识符用来标记常量、变量、数据类型、函数及程序的名字。在C语言中构成标识符必须将合下列语法规则：

1、以字母或下划线“_”中任一字符打头。

2、在第一个字符之后，可以是任意的字母、下划线或数字组成的字符序列，这个序列可以是空串。

C语言中的标识符可以分为下述3类：

1、关键字

关键字是用来说明C语言中某一固定含义的字。float是关键字，它用以说明变量是浮点类型。C语言中一共有如下32个关键字：

int char float double short long unsigned struct union auto extern register static typedef goto return sizeof break continue if else do while

switch case default enum for void const volatile enum

这些关键字为 C 语言专用字，不得赋予其它含义。C 语言中的习惯是用小写字母，所有这些关键字也都是由小写字母构成的。

2、特定字

特定字是具有特定含义的标识符，主要有如下 7 个：

define include undef ifdef ifndef endif line

它们主要用在 C 语言的预处理程序中，这些标识符虽然不是关键字，但由于给这些字赋予了特定含义，所以人们习惯把它们也看作是关键字。因此在程序中不能把这些特定字当作一般标识符使用。

3、一般标识符

通常是用户根据前面所提到标识符构成规则定义的标识符，根据标识符的构成规则，下列用户定义字都是合法的标识符：

Hello, valid, VALID_identifier, Also_valid, _hello, a1, b_2a, _100

用户定义的标识符没有固定含义只有给出说明后才具有特定的含义。在数学中，我们用罗马字母表示常量或变量，在 C 语言中，凡是要命名的对象(如变量、常量、函数、数据类型等)，都用标识符来标识其名字。

C 语言中大小写字母是具有不同的含义的，例如 name 和 NAME 就代表不同的标识符。在选取标识符时不仅要保证正确性，还要考虑容易区分，不易混淆。例如数字 1 和字母 i 在一起，就不易辨认。在取名时，应使名字有很清楚的含义。比如：用标识符 area 表示“面积”，用标识符 length 或 len 表示长度等。对一个可读性好的程序，其中标识符必须选择恰当，取名统一规范，使读者一目了然。

0.0.2 基本数据类型

C 语言提供了四种基本数据类型它们是整型(int)、浮点型（float）、双精度浮点型(double)和字符型(char)。被说明为 int 型的变量可包含整数值。被说明为 float 型和 double 型的变量都可用来存储浮点数(带小数点的数)，只不过 double 型的精度是 float 型的两倍。

char 型的变量都可用来存储一个字符，包括字母、数字字符、标点符号等。

数据有常量和变量之分。常量是指在运算过程中，其值不变的量，可以是一个具体的值，也可以定义一个标识符来代表，即符号常量。变量的值在运算过程可以改变。每个变量都应有一个唯一的名称，并根据其类型在存储器中占一定的存储单元，以便存放其值。

0.0.2.1 数值常量

C 语言中使用的数值常量有 2 种：整数和浮点小数。

1、整数常量

整数常量可以用十进制数、八进制数和十六进制数形式表示。

十进制整数常量由阿拉伯数字 0~9 和负号组成，如 235，64，-5 等。

八进制整数常量以 0 开头，后随若干个阿拉伯数字 0~7 所组成，如 0564，034 分别表示十进制数的 372 和 28。

十六进制整数常量以 0x 开头，后随若干个 0~9 和 A~F 之间的数字所组成，如 0xF 表示十进制数的 15，0x13B 表示十进制数的 315。

2、浮点常量

C 语言中的浮点小数描述的是实数，可以采用十进制小数形式或者指数形式表示。

十进制小数形式包含整数部分、小数点和小数部分。例如：1.45、-34.0、.56、0.0 等，其中需要注意的是小数点不能省略。

指数形式：包含尾数部分、字母 E 或 e 和阶码。例如：1.35E10、1.25e-5 分别表示 1.35×10^{10} 、1.25×10^{-5} 。需要注意的是：尾数部分不能省、阶码必须是整数。

0.0.2.2 字符型常量

C 语言把字符型常量分为两种类型：字符常量和字符串。

1、字符常量

字符常量的表现形式是由单引号括起来的一个字符，例如：

```
'a','B','-','1'
```

其中单引号、双引号和反斜杠的表现形式比较特殊，分别是'\'','\"','\\'，这种字符称为转义字符。在 C 语言中转义字符被认为是具有特殊意义的单个字符。常见的以\开头的专业字符如表 3-1 所示。

表 3-1 常用转义字符表

转义字符	功能
\n	回车换行符，光标移动到下一行行首。
\r	回车不换行，光标移动到本行行首。
\t	横向跳格（8 位为一格，光标跳到下一格起始位置）。
\b	退一格，光标 1 往左移动一格。
\f	走纸换页。
\\	输出反斜杠字符“\”。
\'	输出单引号字符“'”。
\"	输出双引号字符“"”。
\ddd	三位八进制数 ddd 对应的 ASCII 码字符。
\xhh	两位十六进制数 hh 对应的 ASCII 码字符。

2、字符串

字符串是由双引号括起来的字符序列，例如：

```
"Hello world!","C program","120.45"
```

等都是字符串。

不要将字符常量与字符串混为一谈。’a’是字符常量，而“a”是字符串，二者有本质的区别。C 系统自动在每一个字符串的末尾加一个字符串结束标志，系统据此判断字符串是否结束。字符串结束标志是一个 ASCII 码值为 0 的字符，即‘\0’。从 ASCII 码表中可以看到 ASCII 码为 0 的字符是空操作字符．它不引起任何控制动作，也不是一个可显示的字符。如果有一个字符串“CHINA”，则它在内存中占 6 个字节，最后一个字符为‘＼0’。但在输出时不输出‘\0’，所以字符串“CHINA”有效的字符个数是 5。

0.0.2.3 变量

变量的值可以改变，C 语言中所有的变量使用前必须先定义，说明变量类型。对于

每个变量都必须确定其数据类型。定义就是用来规定一组变量的数据类型。

1、变量的定义

变量定义的形式如下：

```
类型标识符 变量名,1 变量名 2, …, 变量名 n;
```

其中，类型标识符用以说明变量的数据类型，对于基本数据类型来说，它可以是 int、float、double 和 char，分别表示整形、浮点型、双精度型和字符型。例如：

```
int i, j, k;
float length;
double height;
char c;
```

分别说明变量 i、j 和 k 是整形的，变量 length 是浮点型的，变量 height 是双精度型的，变量 c 是字符型的。变量名是用户自定义的一般标识符。

变量在定义时要注意以下几个问题：

1）允许在一个类型标识符后，说明多个相同类型的变量。类型标识符与变量名之间至少用一个空格间隔，各变量名之间用逗号间隔，最后一个变量名之后必须以“；”号结尾。

2）变量定义必须放在变量使用之前。一般放在函数体的开头部分。变量名在同一函数内不允许作两次说明。例如，下面函数中对变量 a 的说明显然是不正确的：

```
main()
{
  int a;
  float a;
  …
  }
```

3）变量的命名要符合 C 语言规定的标识符命名规则，即只能由字母、数字和下划线组成，首字符必须为字母或下划线。此外，C 语言中规定的有特殊用途的关键字，如 int、float、if 等，不能作为变量名称。如 a、b3、stu 等是正确的，而 2cd、tea-1、sh#3、if、else、while、float 等是错误的。

4）C 语言中大小写是敏感的，即 int 和 Int 是不同的，Int 不是关键字可以做变量名。

但习惯上，C 中的变量一般用小写字母表示。

5）变量的数据类型决定了它的存储类型，即该变量占用的存储空间。所以定义变量类型就是为了给该变量分配存储空间，以便存放数据。

基本的变量类型及其存储空间如表 3-2 所示。

表 3-2 基本数据类型表

类型	名称	存储空间	取值范围 实例	
int	整型	4 个字节	- 2147483648~ 2147483647 的整数	int i,j;
float	单精度浮点型	4 个字节	实数，有效位数 6~7 位	float x;
double	双精度浮点型	8 个字节	实数，有效位数 15~16 位	double y;
char	字符型	1 个字节	ASCII 码字符，或-128~127 的整数	char c;

变量的类型决定了它可以存放的数据范围，所以在处理数据时，一定要先考虑清楚数据的特征和范围，再确定使用何种类型变量存放数据。

2、变量的赋值

在定义一个变量时，系统自动根据变量类型分配了存储空间。但是当变量的值即存储在该空间的值并未给出时，其值可能是一个内存中的随机值，所以变量需要预置一个值，即赋值。赋值操作通过赋值运算符“=”把右边的值赋给左边的变量：

```
变量名=表达式;
```

其执行过程为：

1） 计算赋值运算符“=”右边表达式的值（表达式在下一节介绍）；

2） 将值赋给左边的变量。

例如：

```
a=3; b=b+2; c=3*5-1;
```

其中需要注意的是：

1）数学中的“=”符号不同于 C 语言中的赋值运算符“=”，在 C 语言中，b=b+2 是成立的，它表示变量 b+2 的值赋给变量 b。在 C 语言中判断 2 个数是否相等时使用符号“==”。

2）如果赋值时“=”两侧的类型不一致，系统将做如下处理：将实数赋给一个整型变量时，如将 1.2 赋给 int 型变量啊，即 a=0.9，系统会自动舍弃小数部分，此时 a=3。

3）将整数赋给一个浮点型变量时，如将 34 赋给 float 型变量 c，即 c=12，系统将保持数值不变并且以浮点小数形式存储到变量中，此时 c=12.000000。

4）当字符型数据赋给一个整型变量时，如将'A'赋给 int 型变量 x，即 x='A'，不同的系统实现的情况不同，一般当该字符的 ASCII 值小于 127 时，系统将整型变量的高字节置 0，低字节存放该字符的 ASCII 值，所以此时的 x=字符 A 的 ASCII 值 65。

例 0.4：下面的程序给出了一个赋值时类型自动转换的示例。

```
#include <stdio.h>
main()
{
 int a,b;
 float c;
 a=1.2;
 c=34;
 b='B';
 printf("a= %d,c=%f ,b=%d\n",a,c,b);
}
```

在上面的程序中，实数 1.2 赋给整型变量 a，舍弃小数保留整数部分。整数 34 赋给浮点型变量 c，转换为等值的浮点小数形式。大写字母 B 的 ASCII 码的十进制形式为 66，小于 127，因此 b=66。程序的运行结果如下：

```
a= 1,c=34.000000 ,b=66
```

变量在定义的同时也可以赋初值，称做变量的初始化。例如：

```
int a,b=0;等价于  int a,b; b=0;
float c=1.1,d=2.2;  等价于 float c,d; c=1.1; d=2.2;
```

字符型变量的值可以是字符型数据、介于-128~127 的整数或者转义字符。

计算机存储的是二进制数，所以将一个字符数据存放到一个字符变量中，实际上存储的是该字符对应的 ASCII 码的二进制形式。例如大写字符 A 的 ASCII 码用二进制表示是 01000001、十进制的标识是 65、八进制的标识是 101、十六进制的标识是 41，即字符型常量'A'、十进制整数 65、转义字符'\101'或'\x41'与二进制码 01000001 相同，因此这四种形式的数据在计算机中的存储形式相同，赋给一个字符型变量的结果也相同。

0.0.2.4 类型修饰符

上述介绍的基本类型可以带修饰性前缀，即类型修饰符，用来适用更多不同特点的数据处理的需要，扩大 C 语言基本数据类型的使用范围。C 语言共有 4 种类型修饰符：long（长型）、short（短型）、signed（有符号型）、unsigned（无符号型）。short 型和 long 型用于整型和字符型，其中 long 型还可以用于双精度型。short 型不常用，对于不同机型取值范围不同，这里不再介绍。long int（简写为 long）型的存储长度为 4 个字节，范围为 $-2^{31} \sim 2^{31}-1$，用于存储整数超过 int 型取值范围的情况。long double 型存储长度 16 个字节，约 24 位有效数字，取值范围超过 double 型的数据。

例如：

```
long int a;
unsigned int b;
short int c;
```

分别表示 a 是长整型，b 是无符号整型，c 是短整型数据。

有符号型 signed 和无符号型 unsigned 适用于 char 型、int 型和 long 型三种类型，区别在于它们的最高位是否作为符号位。unsigned char 型取值范围为 0~255，unsigned int（简写为 unsigned）型取值范围未 0~65535，unsigned long 型取值范围为 $0 \sim 2^{32}-1$。

0.0.3 运算符和表达式

C 语言是一种表达式语言。表达式是由运算符和运算分量构成的，运算符是一个表示对数据进行特定运算的符号，参加运算的数据本身就叫运算分量，运算符对运算分量进行运算。例如：

```
5+4-2
```

是一个表达式，符号“+”和“-”就是运算符，常数 5、4 和 2 即为运算分量，和日常用法一样，符号“+”和“-”分别表示加法和减法，因此，上面这个表达式通知 C 编译程序加或减这些常数。运算符进行运算的运算分量的个数称为运算符的目数，只有一个运算分量的运算符称为单目运算符，有两个运算分量的运算符称为双目运算符，有三个运算分量的运算符称为三目运算符。例如：“+”这个运算符有两个运算分量，故为双目运算符。

0.0.0.1 运算符的种类

C 语言的运算符可分为以下几类：

1、算术运算符

用于各类数值运算。包括加(+)、减(-)、乘(*)、除(/)、求余(或称模运算，%)、自增(++)、自减(--)共 7 种。

2、关系运算符

用于比较运算。包括大于(>)、小于(<)、等于(==)、 大于等于(>=)、小于等于(<=)和不等于(!=)6 种。

3、逻辑运算符

用于逻辑运算。包括与(&&)、或(||)、非(!)3 种。

4、位操作运算符

参与运算的量，按二进制位进行运算。包括位与(&)、位或(|)、位非(~)、位异或(^)、左移(<<)、右移(>>)6 种。

5、赋值运算符

用于赋值运算，分为简单赋值(=)、复合算术赋值(+=,-=,*=,/=,%=)和复合位运算赋值(&=,|=,^=,>>=,<<=)三类共 11 种。

6、条件运算符

这是一个三目运算符，用于条件求值(?:)。

7、逗号运算符

用于把若干表达式组合成一个表达式(，)。

8、指针运算符

用于取内容(*)和取地址(&)2 种运算。

9、求字节数运算符

用于计算数据类型所占的字节数(sizeof)。

10、特殊运算符

有括号()，下标[]，成员(->，.)等几种。

在本节的后面部分小节中，集中介绍算术运算符、逻辑运算符、赋值运算符、条件运算符以及逗号运算符，其它运算符在后面各节中另行介绍。

0.0.0.2 优先级和结合性

C 语言中，运算符的运算优先级共分为 15 级，分别以数 1~15 来代表，数字越大，优先级越高。在表达式中，优先级较高的先于优先级较低的进行运算。 而在一个运算量两侧的运算符优先级相同时， 则按运算符的结合性所规定的结合方向处理。 C 语言中各运算符的结合性分为两种，即左结合性(自左至右)和右结合性(自右至左)。例如算术运算符的结合性是自左至右，即先左后右。如有表达式 x-y+z 则 y 应先与“-”号结合， 执行 x-y 运算，然后再执行+z 的运算。这种自左至右的结合方向就称为“左结合性”。而自右至左的结合方向称为“右结合性”。 最典型的右结合性运算符是赋值运算符。如 x=y=z,由于“=”的右结合性，应先执行 y=z 再执行 x=(y=z)运算。 C 语言运算符中有不少为右结合性，应注意区别，以避免理解错误。表 0.1 列出了各运算符的优先级和结合性。

表 3-1 运算符的优先级和结合性

优先级	运算符	结合性
15	(), [ ], . , ->	右
14	!, ~, ++, --, +, -, &, *(类型名), sizeof	左
13	*, /, %	左
12	+, -	右
11	<<, >>	右
10	<, <=, >, >=	右
9	==, !=	右
8	&	右
7	^	右

6	|	右
5	&&	右
4	||	右
3	?:	左
2	=, *=, /=, %=, +=,-=, >>=, <<=, &=, ^=, |=	左
1	,	右

0.0.0.3 算术运算符和算术表达式

1、算术运算符

基本的算术运算符有：

1、加法运算符“+”，加法运算符为双目运算符，即应有两个量参与加法运算。如 a+b,4+8 等，具有左结合性。

2、减法运算符“-”，减法运算符为双目运算符，具有左结合性。但“-”也可作负值运算符，此时为单目运算，如-x，-5 等，具有右结合性。

3、乘法运算符“*”，乘法运算符为双目运算，具有左结合性。

4、除法运算符“/”，除法运算符为双目运算，具有左结合性。参与运算量均为整型时， 结果也为整型，舍去小数。如果运算量中有一个是实型，则结果为双精度实型。

5、求余运算符（模运算符）“%”，求余运算符为双目运算，具有左结合性。要求参与运算的量均为整型。 求余运算的结果等于两数相除后的余数。

6、自增、自减运算符

自增运算符记为“++”，其功能是使变量的值自增 1。自减运算符记为“--”，其功能是使变量值自减 1。自增、自减 运算符均为单目运算，都具有右结合性。可有以下几种形式：

1）++i：i 自增 1 后再参与其它运算。

2）--i： i 自减 1 后再参与其它运算。

3）i++：i 参与运算后，i 的值再自增 1。

4）i--： i 参与运算后，i 的值再自减 1。

2、算术表达式

算术表达式是由算术运算符和括号连接起来的式子， 以下是算术表达式的例子：

```
a+b   (a*2)/c   (x+r)*8-(a+b)/7   ++i   sin(x)+sin(y)   (++i)-(j++)+(k--)
```

0.0.0.4 关系运算符及表达式

关系运算符有 6 个， “>”、“<”、“==”、“>=”、“<=”、“!=”，分别表示“大于”、“小于”、“等于”和“大于等于”、“小于等于”和“不等于”，，其中，“==”和“!=”的优先级为 9，其它运算符优先级为 10。它们都是双目运算符，都是左结合的，如果两个运算分量不同，要进行类型转换。注意，C 语言中“=”是赋值运算符，而“==”才是真正的等号。表达式的运算分量是基本数据类型，表达式的值是整型，如表达式成立，值为 1，否则为 0。C 语言中，总是用非 0 值表示“真”，用 0 表示“假”。

0.0.0.5 逻辑运算符与逻辑表达式

逻辑运算符主要用于关系表达式所构成的条件的复合，条件的复合从根本上来说有三种：第一种是“两个条件都成立”，这就是两个条件的“与”运算；第二种是“两个条件至少有一个成立”，这就是两个条件的“或”运算；还有一种是“某个条件不成立”，这就是一个条件的“非”运算，它们在 C 语言中分别用运算符“&&”、“||”和“!”表示。

逻辑运算符的运算对象是简单数据类型（除数组和结构外的任何类型），“&&”、“||”是双目运算符，“!”是单目运算符。它们的运算结果是 int 型的，一般说来只有“0”和“1”两个值。对于“&&”，计算时只有当两个运算对象都不为 0 时表达式的值为 1，其他情况表达式的值都为 0；对于“||”，计算时只有当两个运算对象都为 0 时表达式的值为 0，其他情况表达式的值都为 1。

“&&”和“||”还有一个特别的性质，它们是严格从左至右进行运算的，当运算到

一定程度，表达式的值可以被确定时，就不再运算下去。例如，在计算表达式

```
((y=3)>4)&&((y=5)<7)
```

时，表达式(y＝3)的值为 3，显然((y＝3)＞4)的值为 0，对于“&&”，计算时只有当两个运算对象只要一个为 0 时表达式的值即为 0，此时表达式的值已经确定，所以就不对((y=5)＜)7 求值了，执行此表达式后 y 的值为 3 而不为 5。

0.0.0.6 复合赋值符及表达式

在赋值符“=”之前加上其它二目运算符可构成复合赋值符。如+=，-=，*=, / =，%=，<<=，>>=，&=，^=，|=。 构成复合赋值表达式的一般形式为：

```
变量 双目运算符=表达式
```

它等效于：

```
变量=变量 运算符 (表达式)
```

例如： a+=5 等价于 a=a+5；x*=y+7 等价于 x=x*(y+7)； r%=p 等价于 r=r%p。

复合赋值符这种写法，对初学者可能不习惯，但十分有利于编译处理，能提高编译效率并产生质量较高的目标代码。

0.0.0.7 条件运算符与条件表达式

条件运算符“?:”是 C 语言中唯一的一个三目运算符，由它组成的表达式的形式为

表达式 1? 表达式 2: 表达式 3

计算条件表达式时，首先计算表达式 1，如果表达式 1 的值不为 0(即条件为真)，计算表达式 2，条件表达式的值为表达式 2 的值；如果表达式 1 的值为 0(即条件为假)，计算表达式 3，条件表达式的值为表达式 3 的值。条件表达式值的类型总是表达式 2 和表达式 3 的类型中较高的类型。必要时要进行类型转换。

条件运算符的优先级为 3，右结合。

0.0.0.8 逗号运算符和逗号表达式

C 语言中逗号“，”也是一种运算符，称为逗号运算符。 其功能是把两个表达式连接起来组成一个表达式， 称为逗号表达式。其一般形式为：

```
表达式 1, 表达式 2
```

逗号表达式的求值过程是分别求两个表达式的值，并以表达式 2 的值作为整个逗号表达式的值。例如：

```
a=b,c;
```

分别对 b 和 c 求值，最后以 c 的值为整个表达式的值，所以执行上述语句后 a=c。

对于逗号表达式还要说明几点：

1)逗号表达式一般形式中的表达式 1 和表达式 2 也可以又是逗号表达式。例如：表达式 1，(表达式 2，表达式 3) 形成了嵌套情形。因此可以把逗号表达式扩展为以下形式：表达式 1，表达式 2，…表达式 n 整个逗号表达式的值等于表达式 n 的值。

2）程序中使用逗号表达式，通常是要分别求逗号表达式内各表达式的值，并不一定要求整个逗号表达式的值。

3）并不是在所有出现逗号的地方都组成逗号表达式，如在变量说明中，函数参数表中逗号只是用作各变量之间的间隔符。

0.0.4 数据类型转换

C 语言中不同数据类型的取值范围不同，在进行混合运算时需要进行类型转换。

1、自动类型转换

C 语言规定，不同类型的数据在参加运算前会自动转换成相同的类型，再进行运算。转换的规则是：如果运算的数据有 float 型或 double 型，自动转换成 double 型再运算，结果为 double 型。如果运算的数据中无 float 型或 double 型，但是有 long 型，数据自动转换成 long 型再运算，结果为 long 型，其余情况为 int 型。

2、强制类型转换

在 C 语言中也可以使用强制类型转换符，强制表达式的值转换为某一特定类型。强制类型转换形式为：

```
(类型) 表达式
```

强制类型转换最主要的用途之一是满足一些运算对类型的特殊要求，例如求余运算符“%”，要求运算符两侧的数据为整型，如（int）2.5%3；其次是为了防止整数除法中的小数部分丢失，例如：

```
int a=3,b=2;
float c;
c=(float)a/b;
```

此时，c 的值为 1.500000。如果不用强制类型转换，即 c=a/b，结果为 1.000000。

需要注意的是，在强制类型转换时，原来变量的类型并未发生变化，如上例中(float) a 的值为 float 型的 0.0，但是 a 的类型不变，仍然为整型数据 3。

0.4 基本语句与输入输出

C 程序的执行部分是由语句组成的，程序的功能也是由执行语句实现的。本节介绍 C 语言中语句的基本知识和数据的输入（键盘输入）和输出（显示输出）。

0.4.1 C 语言语句概述

C 语言程序是由函数组成的，而函数包含数据定义部分和执行部分，执行部分即由语句组成。和其它高级语言一样，C 语言的语句用来向计算机发出操作命令。程序应该包括数据描述（由数据定义部分来实现）和数据操作（由语句来实现）。数据描述主要定义数据结构（用数据类型表示）和数据初值。数据操作的任务是对已提供的数据进行加工处理。

C 语句可以分为以下五类：

1、控制语句

它可完成一定的控制功能，C 语言中只有 9 种控制语句，它们是：

1） if（）~else~ （条件语句）

2）for（）~ （循环语句）

3）while（）~ （循环语句）

4）do~while（） （循环语句）

5）continue （结束本次循环语句）

6）break （终止执行 switch 或循环语句）

7）switch （多分支选择语句）

8）goto （跳转语句）

9）return （从函数返回语句）

上面 9 种语句中的括号（）表示其中是一个条件，根据该条件的值决定程序的流程，~表示内嵌的语句。例如：

```
if(x>y)   z=x;
else z=y;
```

是一个条件语句。

2、函数调用语句

由一次函数调用加一个分号构成一个语句，例如：

```
printf("Hello, world!\n");
```

3、表达式语句

一个表达式的最后加上一个分号就构成一个表达式语句。如由赋值表达式及其构成的一个赋值语句分别如下：

```
i＝1      （是表达式，不是语句）
i＝l;     （是一个赋值语句）
```

4、空语句

下面是一个空语句：

```
;
```

即只有一个分号的语句，它不进行任何操作。空语句常用作程序执行流程的跳转目标，或循环语句中的循环体（循环体是主语句，表示循环体什么也不做）。

5、复合语句

可以把若干语句用{}括起来构成复合语句。例如

```
{t=x; x=y; y=t;}
```

{}中包含 3 个赋值语句，但用{}括起来后构成复合语句，注意该复合语句作为一个语句使用，经常用于条件语句和循环语句中。还要注意的是{}中的最后一个语句的最后一个分号不能忽略不写。

0.4.2 数据的输出

数据的输出一般以终端显示器(屏幕)为处理对象。即把程序中的原始数据、中间结果或者最后结果送至终端显示器显示输出。与其它高级语言不同，C 语言本身并没有提供输入和输出语句，而是由 C 语言编译系统在标准函数库中定义了一些输入和输出函数，

程序中的键盘输入和显示输出是通过调用库函数实现的。

由于 C 语言编译系统中的“stdio.h”头文件中包含了与标准输入输出有关的变量定义和相应的宏定义，所以，在利用标准库函数进行输入输出时，一般需要用编译预处理命令#include 将头文件“stdio.h”包含到用户源文件中。

下面介绍如何利用标准库函数来进行标准输入输出，先讨论输出部分。

0.4.2.1 字符输出函数 putchar

putchar 函数的作用是向终端显示器输出一个字符，共一般调用形式如下：

```
char a;
putchar (a);
```

被输出对象 a 放在（）中，，作为 putchar 函数的输入，其作用是把输出对象作为一个字符的 ASCII 码值，将该字符输出到屏幕当前光标位置上。输出对象可以是字符型、整型变量或常量。

0.4.2.2 格式输出函数 printf

上述的 putchar 函数只能输出单个字符，而不能输出其它类型的数据。printf 函数是具有格式控制的输出函数，可以用于输出 C 语言中任意类型的数据．而且可以一次同时输出多个相同类型或不同类型的数据。其一般调用形式如下：

```
printf(格式控制，输出表列);
```

该函数括号内包括两部分：

1）格式控制

格式控制是用双引号括起来的字符串，也称为“格式控制转换字符串”，其中包括格式说明和需要原样输出的普通字符。格式说明部分由“%”和格式控制字符组成，如%c、%d 和%f 等，其作用是将要输出对象转换成指定的格式后输出。

printf 函数中使用的格式字符如表 3-3 所示：

表 3-3 printf 函数中使用的格式字符

格式字符	作用
d	以十进制形式输出带符号的整数，输出长整型数据时，使用 ld。

o	以八进制无符号形式输出整数，输出长整型数据时，使用 lo。
x	以十六进制无符号形式输出整数，输出长整型数据时，使用 lx。
u	以十进制无符号形式输出整数，输出长整型数据时，使用 lu。
c	以字符形式输出单个字符。
f	以十进制形式输出单、双精度浮点数(注意输出双精度 double 型数据时，也用格式 f，而不是 d)。
e	以指数(科学记数法)形式输出单、双精度浮点数。
s	输出指定的字符串。

2）输出表列

printf 函数中的输出表列用以指明输出对象，即要输出的数据，它们可以是常量、变量、表达式或函数值。

若有整型变量变量 i=1，j=2 和实型变量 k=0.1，则

```
printf("i=%d, j=%d, k=%f",i,j,k);
```

中的格式控制字符串为"i=%d, j=%d, k=%f"，其中除%和后续的一个字符作为格式控制字符外，其它字符原样输出。若输出遇到一个格式控制字符时，就要依次到输出表列中寻找一个输出对象，在光标当前位置把输出对象按格式控制字符指定的格式输出。因此上述函数的输出结果为 i=1，j=2，k=0.1，注意用％f 输出实型数据时，默认输出 6 位小数。

格式控制字符的选用应与输出对象的数据类型一致，否则不能得到正确结果。格式控制字符的个数一般也应与输出对象的个数相等，若不相等，编译时不会给出错误提示，但是可能出现不正常的输出结果。若格式控制字符的个数少于输出对象的个数，则依次输出格式控制字符个数的输出对象，多余的输出对象无效；若格式控制字符的个数多于输出对象的个数，也依次输出格式控制字符个数的数据，但只有前面的几个（即输出对象个数）数据有效，而后续的输出数据为随机值，是无效的。

若要输出百分号“％”，如要输出

```
1/4＝25%
```

则格式如下：

```
printf("1/4=%d%%",25);
```

其中的％d 用以输出 25，而％％用以输出一个的百分号。

0.4.3 数据的输入

本节中数据的输入以终端键盘为处理对象，即程序接收用户从键盘输入的数据。

0.4.0.1 字符输入函数 getchar

与 putchar 函数相对应，getchar 函数的作用是接收从键盘输入的单个字符。当程序执行到该函数时，将等待用户从键盘输入一个字符，然后再继续后续程序的执行。其一般调用形式如下：

```
char a;
getchar (a);
```

例 0.5：下面给出了使用 getchar 函数的程序。

```
#include "stdio.h"
main()
{
 char a;
 a=getchar();
 putchar(a);
 printf("\n");
}
```

程序执行到 getchar 后将等待用户输入字符，然后 putchar 再将刚才输入的字符显示到屏幕上。

0.4.0.2 格式输入函数 scanf

getchar 函数只能接收从键盘输入的单个字符，与 printf 函数相对应，scanf 函数是具有格式控制的输入函数，可以用于输入 C 语言中任意类型的数据，而且可以一次同时输入多个相同类型或不同类型的数据。共一般调用形式如下：

```
scanf(格式控制，输入地址表列);
```

该函数括号内包括两部分：

1、格式控制

格式控制是用双引号括起来的字符串，也称为“格式控制转换字符串”，其中包括格式说明和需要原样输入的普通字符。格式说明部分中“%”和格式控制字符组成，如%c、%d 和%f 等，共作用是将输入的数据转换成指定的格式后存入到地址表列所指向

的变量中。

scanf 函数中使用的格式字符与 printf 函数中使用的一样。

2、输入地址表列

输入地址表列用以指明键盘输入数据的保存位置，即输入对象，它必须是地址形式，如变量的地址。如果有多个输入对象，则多个输入地址之间用逗号隔开。在 C 语言中变量的地址可以内取地址运算符“＆”得到，如变量 a 的地址可写成&a。关于地址概念的详细介绍，请参见后续“指针”一节。

例 0.6：利用 scanf 函数，从键盘中输入 2 个整数，1 个浮点数和一个字符，代码如下：

```
#include "stdio.h"
main()
{
 int i,j;
 float c;
 char a;
 scanf("i=%d,j=%d,%f,%c",&i,&j,&c,&a);
 printf("i=%d,j=%d, f=%f,a= %c\n",i,j,c,a);
}
```

上述程序执行到 scanf 函数时，需要用户从键盘输入数据，才能继续执行后续语句。程序执行结果如下：

```
i=2,j=3,1.5,c                /*此处为用户输入数据*/
i=2,j=3, f=1.500000,a= c   /*程序输出结果*/
```

在使用 scanf 函数时还应注意以下问题：

1）scanf 函数中的每个输入格式控制符，都必须在输入参数表列中有一个输入对象与之对应，且格式符必须与相应的输入对象类型一致，即把输入数据保存到对应的输入对象中。

2）当格式控制符之间没有任何字符时，在输入数据时，键盘输入的数据之间可以用空格、回车或 Tab 键间隔。如果格式符之间包含其它字符，则应输入与这些字符相同的字符作为间隔，如上例中的“i=2”、“j=3”和“，”，即格式符之间的其它字符必须原样输入，否则不能把数据存入相应的输入对象。

应该注意，在输入字符型数据时，由于任何字符都能作为有效字符输入，因此，不必用空格、回车或 Tab 键作为间隔。

3）可以在格式符的前面指定输入数据所占的宽度，系统能自动按此宽度裁取相应宽度的数据。如：

```
scanf("%3d,%2d",&i,&j);
```

当键盘输入 12345 时，系统能自动地把 123 赋给变量 i，将 345 赋给变量 j。这种方式也可用于字符型数据的输入，但不能用于实型数据的输入。如

```
scanf("%6.3f",&c)
```

是非法的，不能企图通过程盘输入 123456 而使 f 的值为 120.456，即输入实型数据时不能规定精度。

4）需要再次强调的是，用以指明输入对象的输入地址表列必须是地址形式，如要把键盘输入保存到变量中，则要用变量的地址，而不是变量名。如果只写变量名，编译不会出错，但程序执行时，就会出现混乱。请在学习后续章节中的指针等内容后，仔细加以体会。

例 0.7：编写程序，输入长方体的长、宽、高，计算长方体的表面积和体积。

```
#include "stdio.h"
main()
{
 float length,width,height;
 float area,volume;
 printf("Input length,width,height:\n");
 scanf("%f,%f,%f",&length,&width,&height);
 area=2*(length*width+length*height+width*height);     /*计算长方体表面积*/
 volume=length*width*height;                           /*计算长方体体积*/
 printf("%f,%f,%f\n",length,width,height);
 printf("The surface area is %f, volume is %f\n",area,volume);
}
```

执行到 scanf 函数时，需要用户从键盘输入长方体的长、宽、高，然后计算相应的表面积和体积，最后输出输入的长、宽、高以及表面积和体积。

0.5 流程控制

通常的计算机程序总是由若干条语句组成，从执行方式上看，从第一条语句到最后一条语句完全按顺序执行，是简单的顺序结构；若在程序执行过程当中，根据用户的输入或中间结构去执行不同的任务则为选择结构；如果在程序的某处，需要根据某些条件重复的执行某项任务若干次或直到满足或不满足某条件为止，则是循环结构。大多数情况下，程序都不会是简单的顺序结构，而是顺序、选择、循环三种结构的复杂组合。

C 语言中，有一组相关的控制语句，用以实现选择与循环结构，本节进行简单介绍。

0.5.1 条件控制语句

在程序的三种基本结构中，第二种即为选择结构，其基本特点是：程序的流程由多路分支组成，在程序的一次执行过程中，根据不同的情况，只有一条支路被选中执行，而其他分支语句被直接跳过。

C 语言中，提供 if 语句和 switch 语句选择结构，if 语句用于两者选一的情况，而 switch 用于多分支选一的情形。

0.5.1.1 if 语句

if 条件选择语句是用来判定所给定的条件是否满足，根据判定的结果（真或假）决定给出的两种操作之一。

if 语句的一般形式如下：

```
if(表达式)
        语句 1
else
        语句 2
```

if 语句的执行过程是：首先计算表达式的值，若表达式的值为真（即表达式为非零值），则执行语句 1，若表达式的值为假（即表达式的值为零），则执行语句 2。如：

```
if(a==1)
        b=2;
else
        b=0;
```

当 a 等于 1 时，b 的值为 2，否则 b 的值为 0。

在使用 if 语句时，需要注意以下几个问题：

1）if 后的圆括号中除了可以用关系表达式和逻辑表达式作为判断的条件外，还可以是任意类型的表达式，如算术表达式。例如 if(a+ b)是以 a+b 的值是否为 0 作为判断的条件，而 1f(!(a+b))则以 a+b 的值是否为非零作为判断的条件。

2）在 if 语句中可以省略 else 和语句 2 部分，这时 if 语句的形式如下：

```
if(表达式)  语句 1
```

它是上述一般 if 语句的一种特殊情况，执行过程是：当表达式的值为真的，执行语句 1，否则直接转到此 if 语句的下一条语句去执行。

3）if 语句也可用相应的由条件表达式构成的赋值语句来实现。如以下 if 语句

```
if(表达式 1)
   a＝表达式 2;
   else
  a=表达式 3;
```

可以用下面的赋值语句实现：

```
a＝表达式 1? 表达式 2:表达式 3;
```

4）在 if 语句中，语句 1 和语句 2 既可以是单个语句，也可以是出多个语句组成。但当语句 1 和语句 2 是由多个语组成时，需要用花括号{}括起来，构成复合语句，因为在 C 语言中，复合语句当成一个语句来处理。

5）if 语句本身可以嵌套使用。也就是说，if 语句中的语句 1 和语句 2 还可以是 if 语句。如：

```
if(…)
        if(…)
                语句 1
        else
                语句 2
else
        if(…)
                语句 3
        else
                语句 4
```

由于 if 语句中的 else 可以省略，所以，当 if 语句嵌套使用时，会出现 if 与 else 的配

对问题。如：

```
if(…)
        if(…)
                语句 1
else
```

在上面的语句中，有两个 if 和一个 else，显然 else 与不同的 if 配对，则语句的执行效果不同。虽然 eIse 在书写上与第一个 if 对齐，但实际上它不与第一个 if 配对。C 语言规定，else 总是与其前面最近且未配对的 if 配对，而与书写对齐方式无关。根据此规定，上面语句中的 else 与第二个 if 配对，等效于

```
if(…)
{ if(…)
        语句 1
  else
        语句 2
}
```

如果要改变这种默认的配对关系，使得 else 与第一个 if 配对，可以在相应的 if 语句中加上花括号{}来确定新的配对关系，如：

```
if(…)
        {if(…)
                语句 1
}
else
 语句 2
```

6）嵌套的 if 语句有一种特殊形式，用以处理多种不同的情况：

```
if(表达式 1)            语句 1
else if(表达式 2)       语句 2
else if(表达式 3)       语句 3
    …
else if(表达式 i)       语句 i
else                    语句 j
```

当嵌套的 if 语句较多时，程序冗长，结构相对复杂，不明显直观，降低了程序的可读性，可以改用下节介绍的 switch 多分支选择语句来处理。

0.5.1.2 switch 语句

上节介绍的 if 条件选择语句一般适用于两路选择，即在两个分支中选择其中一个分

支执行，虽然可以用嵌套的 if 语句或不嵌套的多个 if 语句来实现多路选择，但这样使得程序冗长，可读性降低。C 语言中的 swithc 多分支选择语句，可更方便地直接用于多分支选挥。

switch 多分支选择语句的一般形式如下：

```
switch(表达式)
{       case  常量表达式 1:  语句 1
  case  常量表达式 2:  语句 2
  case  常量表达式 3:  语句 3
          …
  case  常量表达式 k:  语句 k
  default:     语句 k+1
}
```

switch 语句的执行过程是，当 switch 后面（）中的表达式的值与某一个 case 后面的常量表达式的值相等时，就执行此 case 后面的语句。若所有 case 中的常量表达式的值都没有与该表达式的值匹配的，就执行 default 后面的语句。

需要说明的是：

1）switch 后面（）中的表达式，可以是整型或字符型表达式，也可以是枚举型数据。作为多分支选择判断的条件必须是整型、字符型或枚举型表达式，即其值为整数。

2）每一个 case 后面跟的是一个常量表达式，不能是变量表达式。即 switchh 后面（）中的表达式只能与常量表达式的值进行匹配比较。

3）每一个 case 的常量表达式的值必须互不相同，否则就会出现矛盾的现象（对表达式的值，有多个匹配对象，就有两种或多种执行方案）

4）执行完一个 case 后面的语句后，流程控制转移到下一个 case 继续执行。“case 常量表达式”只是起语句标号作用，并不是在该处进行判断。在执行 switch 语句时，根据 switch 后面表达式的值找到匹配的入口地址，就从此标号开始执行，不再进行判断。因此，应该在执行一个 case 分支后，要使流程跳出 switch 语句的执行，即终止 switch 的执行。可以用一个 break 语句来实现，即在每一个 case 后的语句之后加上 break 语句。

在 case 后面可以包含一个以上语句，但可以不必用花括号括起来作为复合语句，因

为系统会自动执行本 case 后的所有语句。当然，加上花括号构成复合语句也可以。

例 0.8：输入 10 以内的数字，输出对应的英文单词，即 1 输出 one，2 输出 two，3 输出 three …。

```
#include "stdio.h"
main()
{
 int a;
 printf("input integer number(<10): ");
 scanf("%d",&a);
 switch (a){
 case 1:printf("one\n");break;
 case 2:printf("two\n");break;
 case 3:printf("three\n");break;
 case 4:printf("four\n");break;
 case 5:printf("five\n");break;
 case 6:printf("six\n");break;
 case 7:printf("seven\n");break;
 case 8:printf("eight\n");break;
 case 9:printf("nine\n");break;
 default:printf("error\n");
 }
}
```

程序执行结果如下：

```
input integer number(<10): 3   /*输入数字 3*/
three                          /*程序输出结果*/
input integer number(<10): 11      /*输入数字 11*/
error                          /*程序输出结果*/
```

0.5.2 循环控制语句

循环控制结构是程序中的另一个基本结构。循环结构可以使我们只写很少的语句而让计算机反复执行，从而完成大量雷同的计算。

C 语言提供了 for 语句、while 语句、do…while 语句实现循环结构。

0.5.2.1 for 语句

for 语句的一般格式为：

```
for ( 初始表达式; 循环条件; 循环表达式)
    循环体
```

说明：

1）初始表达式，用于循环开始前，为循环变量设置初始值。

2）循环条件是一个逻辑表达式，若该式取值为真，即条件满足，则继续执行循环，否则，执行循环体后的语句。

3）循环表达式定义了每次循环时，循环变量的变化情况。

4）循环体可以是一条语句或一组语句，若是一组语句，需要用{}括起来。

例 0.9：用 for 循环语句打印输出 1~10。

```
#include "stdio.h"
main()
{
 int i;
 for(i=1;i<=10;i++)
        printf("%d\n",i);
}
```

上述程序的执行过程是：

1）计算初始表达式（i=1）。

2）判断循环条件（i<=10?），若满足，则执行；否则，退出循环。

3）执行循环。

4）返回第 2)步。

在整个 for 循环过程中，初始表达式只计算一次，循环条件和循环表达式则可能计算多次。循环体可能多次执行，也可能一次都不执行。

例 0.10：用 for 语句计算 1+2+…+1000。

```
#include <stdio.h>
main()
{
 int i,sum=0;
 for(i=1;i<=1000;i++)
        sum+=i;
 printf("1+2+...+1000= %d\n",sum);
}
```

在 for 循环中，循环变量的值可以带动循环体中。在上面的程序中，sum+=i；等同

于 sum=sum+i；是 C 语言中更简练、更常用的形式，程序执行结果如下：

```
1+2+...+1000= 500500
```

0.5.2.2 while 语句

while 循环语句的一般形式下：

```
while ( 表达式)
  语句
```

其执行过程是：先计算 while 后面圆括号内表达式的值，如果其值为“真”(非 0)，则执行语句部分(即循环体)。然后，再次计算 while 后面圆括号内表达式的值，并重复上述过程，直到表达式的值为“假”(0)时，退出循环，并转向该循环语句下面的后续语句去执行。如上面计算 1+2+…+1000 用 while 语句可表示为：

```
int i=1,sum=0;
while(i<=1000)
{sum+=i;i++;}
```

使用 while 语句时，应注意以下几个问题：

1）while 后面圆括号内的表达式用作循环判断条件，以决定是执行循环体还是退出循环。该表达式可以是比较表达式和逻辑表达式，也可以是其它任意表达式。如 whlle(i=1){…}，表达式 i=1 为赋值表达式，其值为 1，即循环条件永远为“真”，故该循环是死循环。

2）while 语句的特点是先判断表达式的值，然后执行循环体中的语句，即先判断，后执行，用来实现“当”型循环结构。因此，当执行到 while 循环语句时，若表达式的值一开始就为“假”，则循环体一次也不执行。

3）当循环体由多个语句织成时，必须用花括号{}把它们括起来作为一条复合语句，即在 C 语言中总是只把 while()后的第一条语句作为循环体。

例 0.11：用 while 语句计算 10！。

```
#include   <stdio.h>
main()
{
 int i=1,result=1;
 while(i<=10)
 {
```

```
        result*=i;
        i++;
    }
    printf("10!=%d\n",result);
}
```

程序的执行结果如下：

```
10!=3628800
```

0.5.2.3 do-while 语句

do-while 语句的一般形式如下：

```
do
    循环体语句
while (表达式);
```

其执行过程如下：先执行作为循环体的语句，再计算圆括号内表达式的值，若表达式的值为“真”，则再次执行循环体。如此循环，直到表达式的值为“假”时，结束循环，并转到 do-while 语句下面的后续语句去执行。如计算 1+2+…+1000 用 do-while 语句可表示为：

```
int i=1,sum=0;
do
{           sum+=i;
      i++;
}
while(i<=1000);
```

在书写上应该注意，while 后面圆括号中的右因括号“)”的后面还有一个分号“；”，不能遗漏。

do-while 语句的特点是先执行循环体中的语句，然后判断表达式的值，即先执行，后判断，用来实现“直到”型循环结构：因此，循环体语句至少被执行一次，这一点同 while 语句有所区别。其他注意事项同 while 语句在此不在赘述。

例 0.12：用 do-while 语句计算 1+1/2+…+1/1000 的和。

```
#include <stdio.h>
main()
{
 int i=1;
 float sum=0;
 do{
```

```
        sum+=1.0/i;
        i++;
    }
    while(i<=1000);
    printf("1+1/2+...+1/1000= %f\n",sum);
}
```

由于计算中带有浮点数，因此 sum 定义为 float 型。程序执行结果如下：

1+1/2+...+1/1000= 7.485478

0.5.2.4 循环的嵌套

一个循环的循环体中有另一个循环叫循环嵌套。这种嵌套过程可以有很多重。一个循环外面仅包围一层循环叫二重循环；一个循环外面包围两层循环叫三重循环；一个循环外面包围多层循环叫多重循环。

三种循环语句 for、while、do-while 可以互相嵌套自由组合。但要注意的是，各循环必须完整，相互之间绝不允许交叉。

例 0.13：在屏幕上输出 5 行由“*”构成的三角形图案。

```
#include <stdio.h>
main()
{
 int i,j;
 for(i=1;i<=5;i++)
 {
        for(j=1;j<=i;j++)
              printf("*");
        printf("\n");
 }
}
```

上面的程序中包含了 2 重 for 循环，内层循环根据外层循环变量 i 的值，输出每一行对应的“*”，在每次内层循环结束后，外层循环输出换行符，然后继续下一次内层循环，直至输出所有的“*”。程序的执行结果如下：

```
*
**
***
****
*****
```

读者可以自行用 while、do-while 循环嵌套语句实现上面的程序，在此不再赘述。

0.5.2.5 goto、break 和 continue 语句

程序中的语句通常总是按顺序方向， 或按语句功能所定义的方向执行的。如果需要改变程序的正常流向， 可以使用本小节介绍的转移语句。在C 语言中提供了 4 种转移语句：goto,break, continue 和 return。其中的 return 语句只能出现在被调函数中， 用于返回主调函数，我们将在函数一章中具体介绍。本小节介绍前三种转移语句。

1、goto 语句

goto 语句也称为无条件转移语句，其一般格式如下：

```
goto 语句标号;
```

其中语句标号是按标识符规定书写的符号， 放在某一语句行的前面，标号后加冒号“：”。语句标号起标识语句的作用，与 goto 语句配合使用。如：

```
label: i++;
…
goto label;
```

C 语言不限制程序中使用标号的次数，但各标号不得重名。goto 语句的语义是改变程序流向， 转去执行语句标号所标识的语句。

goto 语句通常与条件语句配合使用。可用来实现条件转移， 构成循环，跳出循环体等功能。但是，在现代程序设计中一般不主张使用 goto 语句， 以免造成程序流程的混乱，使理解和调试程序都产生困难。因此现在程序中极少见到 goto 语句的使用。

2、break 语句

前面已经介绍，break 语句能够跳出 switch 语句，转入 switch 语句的下一语句执行。在 while、do-while、for 循环语句中，一般通过对作为循环条件的表达式的检测来决定是否终止执行循环。此外，在循环体中，也可以利用 break 语句来立即终止循环的执行。

break 语句的一般形式如下：

```
break;
```

其执行过程是终止包含它的 switch 或循环语句的执行，即跳出这 2 种语言，而转动下一语句执行。如：

```
int i,sum=1;
for(i=1;;i++)
{
 sum*=i;
 if(i==10) break;
}
```

在此 for 语句中省略了循环条件表达式，即循环条件永远为“真”，利用循环体中的 break 语句来终止该循环语句的执行。

使用 break 语句时应注意以下问题：

1）break 语句只能用于循环语句或 switch 语句中，且一般与 if 语句一起使用。如：

```
if(…) break;
```

该语句一定位于循环体中或 switch 语句中，当 if(…)中的表达式为“真”，则 break 语句跳出的不是 if 语句，而是跳出包含此 if 语句的循环语句或 switch 语句。

2）对于嵌套的循环语句或嵌套的 switch 语句，break 只能跳出（或终止）它所在的那一层循环或 switch 语句，而不能同时跳出多层循环语句或 switch 语句。

3、continue 语句

continue 语句的作用与 break 语句有些类似，也是用来控制程序流程，但它只能出现在循环体中，而不能出现任 switch 语句中。它的作用在于跳过循环体中该 conthue 语句后面的其它语句，并立即开始下一轮循环条件的判定。所以它与 break 语句不同，并不终止整个循环语句的执行。

continue 语句的一般形式如下：

```
continue;
```

其执行过程是：结束本次循环，即跳过循环体中该 continue 语句后面的语句，而立即进行下一次是否继续执行循环的判定。对于 while 和 do-while 语句来说，也就是立即求解作为循环条件的表达式，而对于 for 语句来说，也就是立即求解循环表达式。

例 0.14：输出 100 以内能被 6 整除的整数。

```
#include   <stdio.h>
void main()
{
 int i;
```

```
for(i=6;i<=100;i++)
{
if (i%6!=0)
continue;
printf("%d ",i);
}
printf("\n");
}
```

上述程序利用求余运算符“%”判断能否被 6 整除，若能整除，则输出该整数，否则进行下一次循环。程序执行结果为：

```
6 12 18 24 30 36 42 48 54 60 66 72 78 84 90 96
```

0.6 函数

在 C 语言中，函数是程序的基本组成单位，因此可以很方便地用函数作为程序模块来实现 C 语言程序。

利用函数，不仅可以实现程序的模块化，程序设计得简单和直观，提高了程序的易读性和可维护性，而且还可以把程序中普通用到的一些计算或操作编成通用的函数，以供随时调用，这样可以大大地减轻程序员的代码工作量。

在 C 语言中可从不同的角度对函数分类：

1、 从函数定义的角度看，函数可分为库函数和用户定义函数两种。

1）库函数

由 C 系统提供，用户无须定义，也不必在程序中作类型说明，只需在程序前包含有该函数原型的头文件即可在程序中直接调用。在前面各节的例题中用到 printf、scanf、getchar、putchar 等函数均属此类。

2）用户定义函数

由用户按需要写的函数。对于用户自定义函数， 不仅要在程序中定义函数本身， 而且在主调函数模块中还必须对该被调函数进行类型说明，然后才能使用。

2、C 语言的函数兼有其它语言中的函数和过程两种功能，从这个角度看，又可把函

数分为有返回值函数和无返回值函数两种。

1）有返回值函数

此类函数被调用执行完后将向调用者返回一个执行结果， 称为函数返回值。如数学函数即属于此类函数。 由用户定义的这种要返回函数值的函数，必须在函数定义和函数说明中明确返回值的类型。

2）无返回值函数

此类函数用于完成某项特定的处理任务， 执行完成后不向调用者返回函数值。这类函数类似于其它语言的过程。由于函数无须返回值，用户在定义此类函数时可指定它的返回为“空类型”， 空类型的说明符为“void”。

3、从主调函数和被调函数之间数据传送的角度看又可分为无参函数和有参函数两种。

1）无参函数

函数定义、函数说明及函数调用中均不带参数。 主调函数和被调函数之间不进行参数传送。 此类函数通常用来完成一组指定的功能，可以返回或不返回函数值。

2）有参函数

也称为带参函数。在函数定义及函数说明时都有参数， 称为形式参数(简称为形参)。在函数调用时也必须给出参数， 称为实际参数(简称为实参)。 进行函数调用时，主调函数将把实参的值传送给形参，供被调函数使用。

4、C 语言提供了极为丰富的库函数， 这些库函数如表 3-4 所示。

表 3-4 C 语言提供的库函数

函数	作用
字符类型分类函数	用于对字符按 ASCII 码分类：字母，数字，控制字符，分隔符，大小写字母等。
转换函数	用于字符或字符串的转换；在字符量和各类数字量 (整型， 实型等)之间进行转换；在大、小写之间进行转换。
目录路径函数	用于文件目录和路径操作。

诊断函数	用于内部错误检测。
图形函数	用于屏幕管理和各种图形功能。
输入输出函数	用于完成输入输出功能。
字符串函数	用于字符串操作和处理。
内存管理函数	用于内存管理。
数学函数	用于数学函数计算。
日期和时间函数	用于日期，时间转换操作。
进程控制函数	用于进程管理和控制。
其它函数	用于其它各种功能。

以上各类函数不仅数量多，而且有的还需要硬件知识才会使用，因此要想全部掌握则需要一个较长的学习过程。 应首先掌握一些最基本、 最常用的函数，再逐步深入。由于篇幅关系，本书只介绍了很少一部分库函数， 其余部分读者可根据需要查阅有关手册。

0.6.1 函数定义与返回值

C 语言虽然提供了丰富的库函数，但这些函数是面向所有用户的，不可能满足用户的各种特殊需要，因此大量的函数必须由用户自己来编写。本节将向读者初步介绍如何定义自己的函数，如何向函数传递简单类型的数据。

0.6.1.1 函数定义

C 语言函数定义的一般形式是：

```
类型说明符  函数名 (数据类型 形式参数 1, 数据类型 形式参数 2, …, 形式参数 n)
{
 说明部分                    /*函数体*/
 语句部分
}
```

类型说明符定义了函数中返回语句返回值的类型，函数返回值不能是数组，也不能是函数，除此之外任何合法的数据类型都可以是函数的类型。在很多情况下都不要求无参函数有返回值， 此时函数类型符可以写为 void。当一个函数没有明确说明类型时， C 语言的编译程序自动将整型（int）作为这个函数的缺省类型，缺省类型适用于很大一部分函数。当有必要返回其它类型数据时，需要分两步处理：首先，必须给函数以明确的

类型说明符；其次，函数类型的说明必须处于对它的首次调用之前。只有这样，C 编译程序才能为返回非整型的值的函数生成正确代码。

函数名是用户自定义的标识符，是 C 语言函数定义中惟一不可省略的部分，需符合 C 语言对标识符的规定，即由字母，数字或下划线组成，用于标识函数，并用该标识符调用函数。另外，函数名本身也有值，它代表了该函数的入口地址，使用指针调用该函数时，将用到此功能。

形式参数用于主调函数利被调函数之间进行数据的传递。需要对形式参数进行类型说明。形式参数表可以是空的，表示该函数为无参函数，即调用该函数时，不需要参数；形式参数表也可以由逗号分开的多个形式参数组成，表示该函数为有参函数，即调用该函数时，需要为其提供参数。形式参数表说明可以有两种表示形式：

```
类型说明符  函数名 (数据类型 形式参数 1, 数据类型 形式参数 2, …, 数据类型 形式参数 n)
{ … }
```

或：

```
类型说明符  函数名 (形式参数 1,形式参数 2, …, 形式参数 n)
  数据类型 形式参数 1;
  数据类型 形式参数 2;
  …
  数据类型 形式参数 n;
{ … }
```

不管形式参数表中是否有参数，都要用左、右圆括号括起来。形式参数可以是变量、数组、指针等，但不能是表达式或常量，因为形式参数用来存放从主调函数传递的实参的值，需要有相应的存储空间。

在函数体的说明部分主要对本函数中用到的有关变量进行定义，说明变量的数据类型，以便后面的语句正确使用变量。函数内定义的变量不可以与形参同名。函数体的语句部分完成函数应的规定的运算，执行的规定的动作，集中体现函数的功能。

应该指出的是，在 C 语言中，所有的函数定义，包括主函数 main 在内，都是平行的，函数定义的位置必须放在所有函数的外部。也就是说，在一个函数的函数体内， 不能再定义另一个函数， 即不能嵌套定义。但是函数之间允许相互调用，也允许嵌套调用。

习惯上把调用者称为主调函数。 函数还可以自己调用自己，称为递归调用。main 函数是主函数，它可以调用其它函数，而不允许被其它函数调用。 因此，C 程序的执行总是从 main 函数开始， 完成对其它函数的调用后再返回到 main 函数，最后由 main 函数结束整个程序。一个 C 源程序必须有，也只能有一个主函数 main。

0.6.1.2 返回语句

C 语言中的返回语句有下列形式：

```
return 表达式;
```

返回语句的作用有两个，一是把控制返回给调用者(函数)，另一是向调用者传递有关信息。例如：

```
retrun;
return -1;
retrun x+y;
retrun ++x;
```

上例中第一个语句没有返回任何值，只是把控制返回给了调用者，而其余的既返回了控制，又返回了值。

一个函数中的返回语句可以有一个、多个或没有，若在执行一个函数时没遇到 return 语句，则从这个函数的末尾一个语句后返回，或者可以认为函数体大括号“}”前有一个无形的 return 语句(不带返回值)。

所有非空值的函数都会返回一个值。我们编写的程序中大部分函数属于三种类型。第一种类型是简单计算型－函数设计成对变量进行运算，并且返回计算值。计算型函数实际上是一个“纯”函数，例如 sqrt 函数（开方运算）和 sin 函数（正弦运算）。第二类函数处理信息，并且返回一个值，仅以此表示处理的成功或失败。例如 write 函数，用于向磁盘文件写信息。如果写操作成功了， write 函数返回写入的字节数，当函数返回- 1 时，标志写操作失败。最后一类函数没有明确的返回值。实际上这类函数是严格的过程型函数，不产生值。

对于具有返回值的函数，调用者可以使用这个值，也可以不使用这个值。例如程序段：

```
getchar();
c=getchar();
```

其中第一个 getchar 执行一次就得到一个字据，但程序不对这个返回值做任何处理，而把接下去第二个 getchar 得到的字符赋给变量 c。

例 0.15：编写求 1+2+…+n 的函数。

```
double fsum (int n)
{
        int i;
        double tempsum=0;
        if(n<=0)
         {
              printf("Input number is error! %d<=0\n",n);
              return -1.0;
         }
         for(i=1;i<=n;i++)
         tempsum+=i;
         return tempsum;
}
```

在上面的程序中，double fsum(int n)是函数的首部，其中 fsum 是函数名，在它前面的 double 是类型名，用来说明函数返回值的类型是双精度型。在本例中只有一个形式参数 n，其类型为 int。

函数体中，除形式参数外，用到的其它变量必须在说明部分进行定义，这些变量(包括形式参数)，只在函数被调用时才临时开辟存储单元、当退出函数时，这些临时开辟的存储单元全被释放掉，因此，这种变量只在函数体内部起作用，与其它函数体中的变量互不相关，它们可以和其它函数中的变量同名。函数体中的说明部分，就像在 main 函数中那样，总是放在函数体中所有可执行语句之前。fsum 函数中定义了一个 int 类型的变量 i、一个 double 类型的变量 tempsum，当退出 fsum 函数后，这些变量，包括形式参数 n 所占的存储单元都不再保留。

fsum 函数中有两处出现 return 语句，函数中通过 for 循环进行连加，求出 1+2+…+n，并把结果放在变量 tempsum 中，最后的 return tempsum 语句，使流程返回到调用该函数的地方，并把所求得的的值带回，fsum 函数的 if 语句中还有一个语句 return -1.0；，在

此表示如果 n 的值小于 0，则不能求连加，在退出函数时同时返回-1.0 作为出错的标志。

0.6.2 函数的调用

函数的一般调用形式为：

```
函数名(实在参数 1, 实在参数 2, …, 实在参数 n)
```

其中，实在参数是函数调用时，主调函数为被调函数提供的原始数据。若实在参数(简称实参)的个数多于一个时，各实在参数之间用逗号隔开。若函数无形参，调用形式为：

```
函数名()
```

函数名后的一对园括号不可少。

可用两种方式调用函数：

1）当所调用的函数用于求出某个值时，函数的调用可作为表达式出现在允许表达式出现的任何地方。例如对于以上的 fsum 函数，若 x 为 double 变量，可用以下语句调用 fsum 函数，求出 10 的连加：

```
x=fsum(10);
```

fsum 函数也可以出现在 if 语句中作为进行判断的表达式：

```
if(fsum(n)<0) …
```

2）函数的调用也可作为一条独立的语句，如

```
函数名(实在参数表);
```

注意最后有一个分号。

函数调用时的语法要求如下：

1）调用函数时，函数名必须与所调用的函数名字完全一致。

2）实在参数的个数必须与形式参数的个数一致，实在参数可以是表达式，在类型上应按位置与形式参数一一对应匹配。如果类型不匹配，C 编译程序按赋值兼容的规则进行转换，若实在参数和形式参数的类型不赋值兼容，通常并不给出出错信息，且程序仍然执行，只是不会得到正确的结果。因此对于初学者来说应该特别注意实在参数和形式参数的类型匹配。

3）C 语言规定：函数必须先定义，后调用(函数的返回值类型为 im 或 char 时除外)。

如，如果想在 main 函数中调用 0.6.1 节的 fsum 函数、在源程序中它们的位置应该如下：

```
double fsum(int n)
{ … }
main()
{
 int a;
 double x;
 …
 x=fsum(a);
 …
}
```

如果被调用函数的返回值为 int 或 char 类型，则被调用函数的定义也可以放在调用的位置之后，例如:

```
main()
{
 …
 f1();
 …
}
f1() {…}
```

4）C 程序中，函数可以直接或间接地自己调用自己，称为递归调用。递归是以自身定义的过程。也可称为“循环定义”。

例 1：利用递归方法计算 n!。

```
factor(int n)   /*  递归调用方法* /
{
   int answer;
   if (n==1)
   return (1);
   answer=factor(n-1)*n;/*  函数自身调用* /
   retrun answer;
}
```

当用参数 1 调用 factor( )时，函数返回 1；除此之外的其它值调用将返回 factor(n-1) * n 这个乘积。为了求出这个表达式的值，用（n-1）调用 factor()一直到 n 等于 1，调用开始返回。

0.6.3 调用函数和被调用函数之间的数据传递

C 语言中，调用函数和被调用函数之间的数据可以通过三种方式进行传递：

1）实在参数和形式参数之间进行数据传递。

2）通过 return 语句把函数值返回调用函数。

3）通过全局变量(有关全局变量的内容将在下一节介绍)。

在 C 语言中，数据只能从实在参数单向传递给形式参数，称为‘按值”传递。也即是说，当简单变量作为实在参数时，用户不能在函数中改变对应实在参数的值。

例 0.16：以下程序说明了函数参数之间的单向传递。

```
main()
{
 int a=1;
 printf("a1=%d\n",a);
 fadd(a);
 printf("a4=%d\n",a);
}

fadd(int n)
{
 printf("a2=%d\n",n);
 n+=3;
 printf("a3=%d\n",n);
}
```

程序运行结果如下：

```
a1=1
a2=1
a3=4
a4=1
```

当程序从 main 函数开始运行时，按定义在内存中开辟了 1 个 int 类型的存储单元 a 且赋初值为 1，fadd 函数调用之前的 printf 语句输出结果验证了 a 的值。当调用 fadd 函数之后，程序的流程转向 fadd 函数，这时系统为 fadd 函数的形参 n 分配了另外 1 个临时的存储单元，同时把实参 a 的值传送给对应的形参 n，因此实参和形参虽然同名，但它们却占用不同的存储单元。

当进入 fadd 函数后，首先执行一条 printf 语句，输出 fadd 函数中的 n 的值，因为未对它们进行任何操作，故仍输出 1，当执行了赋值语句 n+=3；之后，接着又执行一条 printf 语句，再一次输出 fadd 函数中的 n 的值，输出结果为 4；fadd 函数执行完后，这时 fadd

函数中的 n 变量将消失，并且流程返回到 main 函数；最后执行 main 函数中最后一条 printf 语句，输出了 a 的值，由输出结果可见 main 函数中的 a 的值在调用 fadd 函数后没有任何变化。

以上程序运行的结果证实了在调用函数时，实参的值将传送给对应的形参，但形参值的变化不会影响对应的实参。

0.6.4 变量的作用域

C 程序结构是模块式的。即一个 C 源程序是由一个主函数和零个、一个或多个其他函数组成，那么编写函数时，如何命名变量?不同函数中定义的变量是否允许同名?不同的函数能否共享同一组数据？这些问题必须从“变量的作用域”角度来解答。所谓变量的作用域就是指变量能被有效引用的范围。据此，C 语言将变量分为“局部变量”和“全局变量”。

1、局部变量

在函数(包括 main 函数)内部或复合语句内部定义的一切变量(包括形式参数)都是局部变量，它们的作用域仅限于自身所在的函数内部或复合语句内部，出界便无效。因此在不同函数中定义的变量即便同名，仍代表不同的对象。

2、全局变量

在函数外部定义的变量称为“全局变量”。全局变量的作用域为从定义变量的位置开始直至程序结束。因此，全局变量能够供其后面的一切函数引用。

例 0.17：下面的程序说明了全局变量的例子。

```
int i=1;
f1()
{
 i++;
}
main()
{
  printf("i1=%d\n",i);
 f1();
```

```
    printf("i2=%d\n",i);
```

程序的运行结果为：

```
i1=1
i2=2
```

因为主函数和 f1 函数中均未定义名字为 i 的变量。所以在它们的函数体中出现的变量名 i 均为程序首部定义的全局变量，在调用 f1 函数之前，i 的值为 1，调用之后，i 的值已被修改为 2，所以 i 在主函数中的输出值为第 1 次为 1，第 2 次为 2。

0.6.5 变量的生存期

从变量的作用域原则出发，可以将变量分为全局变量和局部变量；换一个方式，从变量的生存期来分，可将变量分为动态存储变量及静态存储变量。

1、动态存储变量

动态存储变量可以是函数的形式参数、局部变量、函数调用时的现场保护和返回地址。这些动态存储变量在函数调用时分配存储空间，函数结束时释放存储空间。动态存储变量的定义形式为在变量定义的前面加上关键字“auto”，如：

```
    auto int a,b,c;
```

“auto”也可以省略不写。事实上，我们已经使用的变量均为省略了关键字“auto”的动态存储变量。有时为了提高速度，可以将局部的动态存储变量定义为寄存器型的变量，定义的形式为在变量的前面加关键字“register”，例如：

```
    register float a,b,c;
```

这样一来的好处是：将变量的值无需存入内存，而只需保存在 C P U 内的寄存器中，以使速度大大提高。由于 CPU 内的寄存器数量是有限的，不可能为某个变量长期占用。因此，一些操作系统对寄存器的使用做了数量的限制。或多或少，或根本不提供，用自动变量来替代。

2、静态存储变量

在编译时分配存储空间的变量称为静态存储变量，其定义形式为在变量定义的前面加上关键字“static”，例如：

```
    static int i=10;
```

定义的静态存储变量无论是做全程量或是局部变量，其定义和初始化在程序编译时进行。作为局部变量，调用函数结束时，静态存储变量不消失并且保留原值。

例 0.18：静态存储变量的使用

```
main( )
{
    inf f( );
    int j;
    for (j=1; j<=3; j++)
    printf ("%d\n"，f( ))；
}
 int f( )
{
    static int i=0;
    i + + ;
    return i;
}
```

程序运行结果：

```
1
2
3
```

从上述程序看，函数 f 被三次调用，由于局部变量 x 是静态存储变量，它是在编译时分配存储空间，故每次调用函数 f 时，变量 x 不再重新初始化，保留加 1 后的值，得到上面的输出。

0.7 编译预处理

在 C 语言中，我们常常会遇到以“#”号开头的语句，如常量定义语句#define PI 0.14159 及文件包含语句#include <stdio.h>，这类语句统称为编译预处理命令。

所谓“编译预处理”就是 C 编译程序对 C 源程序进行编译前，由编译顶处理程序对这类编译预处理命令进行处理的过程。将预处理结果和源程序一起再进行通常的编译处理，就得到目标代码。

C 语言的预处理命令有：#define，#undef，#include，#if，#else 等。

按其功能共分为以下三类：

1）宏定义

2）文件包含

3）条件编译

为了与一般 C 语言语句相区别，这些命令必须在一行的开头以“#”号开始，末尾不再加“；”号，以区别于 C 语言的其他语句。这些由预处理命令组成的预处理命令行的语法与 C 语言中其他部分的语法无关，它们可以根据需要出现在程序的开始及结束部分，其作用一直持续到源文件的末尾，或其他结束其作用的语句出现的位置。本章下面将重点介绍#define、#include 等常用命令的应用。

0.7.1 宏替换

0.7.1.1 不带参数的宏定义

不带参数的宏定义命令行形式如下：

```
#define  宏名  替换文本
```

在 define、“宏名”和“替换文本”之间用空格隔开。例如：

```
#define  LENGTH  300
```

其中标识符 LENGTH 称为“宏名”，是用户自定义的标识符，不能与同在一段程序中的其他标识符同名。编译时，编译预处理程序对源程序中所有名为 LENGTH 的标识符用 300 来替换，这个替换过程称为宏展开。

例 0.19：不带参数的宏定义。

```
#define  PI 0.14159
main()
{
 float r=2.1area;
 area=PI*r*r;
 printf("The Area is %f\n",area);
}
```

上面的程序是求一个圆的面积，在程序的第一行定义了名为 PI 的宏，圆的半径 r=2.1。以上程序的运行结果是：

```
The Area is 10.854412
```

从以上例子可以看到：

1）“宏名”一般用大写字母表示．以示与普通变量名的区别，当然这井非语法上的规定，也可以用小写字母。如：

```
#define pi 0.14159。
```

2）使用宏名替代一个字符串，其中一个主要的目的是减少程序中重复书写某些字符串的工作量，比如在程序中一些不太好记忆的参数，重复书写容易出错且很繁琐，这时用宏名来代替该字符串就可以使程序简单明了。其次当程序的一些常量需要改变时，如果没有宏名，那么整个程序用到该常量的地方都需人工一一修改，若用宏名，则只需改变宏定义命令行，一改全改。如在程序开头有以下宏定义：

```
#define NUM 1000
```

若需要更改 NUM 参数的值为 100，则只需将#define 语句改为：

```
#define NUM l00
```

这样就提高了程序的可移植性。

3）同一个宏名不能重复定义。如：

```
#define PI 0.14159
#define PI 0.1416
```

二条语句不能同时出现在一段程序中。

4）宏定义是用宏名代替一个字符串，也就是简单的置换，并不作语法检查。如：

```
#define PI 0.14159
```

中的 0.14159 如果输入时不小心将数字 1 误输为字母 i，那么在替换时也照样代入，只有在对已作宏展开后的源程序编译链接时才会出现错误。

5）宏定义与一般 C 语言语句不同，末尾不能加分号，否则会连分号一起置换。

6）可以用林#undef 命令终止宏定义的作用域。一般而言，一个宏定义，从被定义开始直至文件末尾，全程有效。若要更改其作用域则可以用#undef 命令。如：

```
#define   AAA   10
main()
{
 ...
}
#undef AAA
```

在#undef AAA 语句后，AAA 的作用域终止，不能再使用未定义的标识符 AAA。

0.7.1.2 带参数的宏定义

在 C 语言程序设计中，宏定义还可以用于带参数的宏。带参数的宏定义的一般形式为：

```
#define 宏名(参数表) 字符串
```

“字符串”中包含“参数表”中所指定的参数。

例 0.20：带参数的宏的定义

```
#define   cube(x)   (x*x*x)
main()
{
 double f;
 float a=2.5;
 f=cube(a);
 printf("f=%f\n",f);
}
```

运行结果为：

```
f=15.625000
```

上面这段程序里“f=cube(a);”语句的含义是用宏定义语句中的字符串(x*x*x)代替cube(x)这个宏名，即等价于 f=a*a*a；而且在替换过程中用实际参数 a 代替宏定义中的参数 x，这有点类似于函数定义时的形式参数在函数被调用时被实际参数替换一样。但是有一点不同，函数的形式参数与实际参数必须是预先定义好的同一种类型，即如果 cube(x)是一个函数，那么形式参数 x 有——个预先定义好的类型，在后面的程序调用语句 cube(a)中 a 必须是与 x 同样的类型，否则就会出错。然而与函数相比，带参数的宏定义就不受此限制，即带参数的宏对参数的类型不敏感。所以，有时可以利用带参数的宏这一特点来写一些简单而常用的函数。

从形式上，带参数的宏定义和函数似乎容易混淆，从上面的例子可以看到，二者在程序中起的作用、以及书写规范上，确实存在一定的相似之处，然而，二者并不相同，主要有以下区别：

1）函数在调用时，先求出实参表达式的值，再用这个值赋给形式参数；但带参数的宏定义仅仅进行字符替换。

2）函数调用是在程序运行时分配临时内存单元；而宏展开是在编译时进行，展开时不分配内存单元，不进行值传递。

3）宏替换不占运行时间，只占编译时间；函数调用占运行时间。

4）宏替换后使源程序增长；而函数调用不会使源程序增长。

5）在使用带参数的宏的时候，对带参数的宏的展开只是用语句中宏名后的实参字符串代替#define 命令中的形参，这与函数参数传递时，实参将值传递给形参不同，在编写带参数的宏时需加以注意。

0.7.2 文件包含

在编写 C 语言程序时，可以把宏定义语句按照功能不同分别存人不同的文件，当需要某一类宏定义时，就无需在程序中重新定义，只要把这些宏定义所在的文件，包含在程序的开头就可以了。

文件包含就是在一个文件中，包含另外一个文件的全部内容，用#include 命令来实现文件包含的功能。

include 命令的形式如下：

```
#include "文件名"或 #include <文件名>
```

如果文件名是用双引号括起来的，系统将先在源程序所在的目录内查找指定的包含文件，如果找不到，再按系统指定的标准方式到相关的目录去寻找。如果文件名用尖括号括起来，则系统将直接按照指定的标准方式到相关的目录中去查找。在预编译时，预编译程序将用指定文件中的内容来替换此命令行。

文件包含在使用时应注意：

1）一条文件包含命令只能包含一个文件。如果需要包含多个文件，就必须使用多条文件包含命令。

例如，如果要包含两个文件 file1.h 和 file2.h 就不能这样写：

```
#include "file1.h,file2.h"
```

或

```
#include "file1.h" ,"file2.h"
```

必须写成：

```
#include "file1.h"
#include "file2.h"
```

其先后顺序与文件内容无关。但如果一个文件要使用另一个文件的内容，则后者文件应写在前面，即如果 file1.h 要使用 file2.h 文件中的内容，则文件包含语句必须按如下顺序写：

```
#include "file2.h"
#include "file1.h"
```

2）文件包含的定义是可以嵌套的，文件包含的嵌套是指一个被包含的文件中还可以包含其他文件。

3）被包含的文件常以.h 结尾，这类文件中往往是程序所需要的一些说明、定义，如符号常量的定义、类型定义、带参数的宏定义、数组、结构、共用体和枚举的定义等等，以及外部变量的定义、函数的定义或说明。但是被包含的文件可以是任意的文件，不一定是.h 文件，也可以是 C 语言源文件。

0.7.3 条件编译

条件编译是指编译时对源程序的某种控制。在一定的条件下，源程序中的某些特殊语句参加编译，而在另一种条件下，同样的这些语句不参加编译，也即 C 语言的语句是否有效要根据是否满足条件编译语句中设置的条件来决定，这就称为条件编译。

0.7.0.1 常用命令及其格式

1）#ifdef 命令

```
#ifdef 标识符
    程序段 1
#else
    程序段 2
#endif
```

其功能是：当“标识符”已经被定义过(通常是用#define 命令定义)，则对“程序段 l”进行编译，否则编译“程序段 2”。其中“#else”部分根据具体要求可以省略，即也可以为如下形式：

```
#ifdef 标识符
#endif
```

2）#ifndef 命令

```
#ifndef 标识符
    程序段 1
#else
    程序段 2
#endif
```

其功能是：若“标识符”未被定义，则编译“程序段”，否则编译“程序段 2”。正好与第一种形式的作用相反。

3）#if 命令

```
#if 表达式
    程序段 1
#else
    程序段 2
#endif
```

该格式中“#if 表达式”语句与前两种格式不同，if 是关键字，“表达式”是常量表达式。其功能是：当“表达式”的值非 0 时．“程序段 1”参加编译，否则“程序段 2”参加编译。当然，根据需要，也可以没有 else 语句，即简化为如下格式：

```
#if 表达式
    程序段
#endif
```

0.7.0.2 条件编译命令的应用

使用条件编译命令可以对源程序的内容进行选择性的编译,比如在调试程序的时候，通常希望输出一些调试用的情息，而在调试完毕后，就不再需要输出这些信息了。

在学习编译预处理命令前,有一个解决方法是在调试程序的时候增加一些输出语句，而在调试完毕，生成最终需要的可执行文件时，再将源程序改动，删去这些语句，这种作法比较麻烦，在删除的过程中容易误删程序本身需要的语句，或少删了本来作为调试用的的语句，影响最终得到的可执行文件的运行与输出效果。另外，如果这段程序再需要调试时，又得把那些凋试语句逐一加上去。掌握条件编译语句后，可以用这类命令来辅助程序调试。

例如，在程序开头定义一个符号常数 DEBUG(值为 l 或 0)，在程序任何需要设置调试信息之处，加入以下信息：

```
#if DEBUG
    调试代码
#endif
```

这里定义 DEBUG 为 1 时，为调试状态，即输出跟踪信息。在调试时，程序中定义 DEBUG 的语句为：

```
#define DEBUG 1
```

而调试结束后，将这条语句改为：

```
#define DEBUG 0
```

那么凡是加入

```
#if DEBUG
    调试代码
#endif
```

的地方，调试代码便不再起作用。

这样，只需修改#define DEBUG 这一条语句，便达到了自动删除或增加程序中众多调试语句的效果。

0.7.4 预定义宏

如前所述，可以用#define 定义程序需要的符号常量即自定义宏，但是下列标识符不可以出现在辑#define 与#undef 命令中：

```
__STDC__
__FILE__
__LINE__
__DATE__
__TIME__
```

这是由于上述标识符是 C 编译器预先定义好的宏，当预处理器遇到这样的宏时，同样将其转换为相应的代码。如果实现是标准的，则宏__STDC__含有十进制常量 1。如果它含有任何其它数，则实现是非标准的。__FILE__表示源文件的名字。__LINE__表示源文件当前的行号。__DATE__宏指令含有形式为月/日/年的串，表示源程序被预编译的日期，源程序编译到目标代码的时间作为串包含在__TIME__中。串形式为时：分：秒。

可以在程序中使用这些已经定义好的宏名，正如使用我们自己定义的宏一样，例如：

```
printf("Today is %s\n",__DATE__);
```

执行上述语句就可以输出程序编译时当前系统日期。

0.7.5 运算符#和##

在 ANSI　C 中，有两个仅用于宏替换的运算符：#和##。

在带参数的宏替换中，如果有带#的形参，那么在宏扩展时实参应替换该#与形参，且将实参加上双引号，变为实际的字符串变量，即#形参被“实参”代替。如包含下列宏定义的源程序：

```
#define STRING(i)　#i
…
printf(STRING(example\n));
```

将被替换为：

```
printf("example\n");
```

即形参#i 被实参 example\n 加上双引号变为“example\n”替换。

而当另一个宏替换的运算持号##出现在宏替换中时，它起的作用是：##被删除，它连接的前后两个词法单位会合而为一，如果##附近有空格，也将会披消除，这种方式通常用来构造标识符。如：

```
#define LINKSTRING(str1,str2) str1##str2
```

则 LINKSTRING(file,name)将在宏替换时变成 filename。

0.8 数组

迄今为止，我们使用的数据类型都属于基本类型，其基本持点是这些数据类型的变量是彼此独立的。为了描述现实问题中的各种复杂的数据，C 语言还提供了由基本数据类型按一定规律组成的结构数据类型。数组是一种结构数据类型，本节介绍它的定义与应用。

0.8.1 数组的概念

在许多应用中，需要存储和处理大量数据。在这今涉及的问题中，我们能够利用少

量的存储单元，处理大量的数据。这是因为我们能够处理每一个单独的数据项，然后再重复使用存储该数据项的存储单元。例如求一个班学生的平均成绩，每个成绩被存储在一个存储单元中，完成对该成绩的处理。在读入下一个成绩时，原来的成绩消失。这种办法允许处理大量成绩，而不必为每一个成绩分配单独的存储单元。然而，一旦某个成绩被处理，在后面就不能再重新使用它了。

在有些应用中，为了其后的处理，需要保存数据项。例如，要计算和打印一个班学生的平均成绩以及每个成绩与平均成绩的差。在这种情况下，在计算每个差之前，必须先算出平均成绩。因此，必须能够两次考查学生成绩。首先计算平均成绩，然后计算每个成绩与这个平均成绩的差。我们不愿意两次输入学生成绩，我们希望在第一步时，将每个学生的成绩保存于单独的存储单元中，以便在第二步时重新使用它们。在输入数据项时，用不同的名字引用每一个存储单元将是烦琐的。如果有 100 个成绩要处理，将需要一个长的输入语句，其中每个变量名被列出一次。也需要 100 个赋值语句，以便计算每个成绩与平均成绩的差。

数组的使用将简化大批量数据的存储和处理。具有相同类型的一批数据看成是一个整体，叫做数组。给数组取一个名字叫数组名。所以数组名代表一批数据，而以前使用的简单变量代表一个数据。数组中的每一个数据称为数组元素，它可通过顺序号（下标）来区分。

例如，一个班 60 名学生的成绩组成一个数组 a，每个学生的成绩分别表示为：

```
a[0],a[1],a[2],…,a[i],a[59]
```

又如某集团 5 个公司全年销售收入组成数组 b，每个公司每月的销售收入分别表示为：

```
b[0][0],b[0][1],b[0][2],…,b[0][11]
b[1][0],b[1][1],b[1][2],…,b[1][11]
…
b[4][0],b[4][1],b[4][2],…,b[4][11]
```

在上例中，区分 a 数组的元素需要一个顺序号，故称为一维数组，而区分 b 数组的元素需要两个顺序号，故称为二维数组。

引入数组的概念后，可以用循环语句控制下标的变化，利用单个语句，就可输入各个数据项。例如，输入 90 名学生的成绩，可描述为：

```
for(i=0;i<90;i++)
scanf("%d",&a[i]);
```

一旦各个数据项存于数组中，将能随时引用任一数据项，而不必重新输入该数据项。

0.8.2 数组的定义和使用

同变量在使用之前要定义一样，数组在使用之前也要定义，即确定数组的名字、类型、大小和维数。本节先讨论一维数组的定义与应用，然后讨论二维数组。

0.8.2.1 一维数组

1、一维数组的定义

一维数组的定义形式为：

```
类型标识符 数组名[常量表达式];
```

其中，类型标识符指明数组元素的类型。数组名是用户自定义的标识符。方括号中的常量表达式的值表示数组元素的个数。常量表达式中可以包括整数常量和符号常量以及由它们组成的常量表达式，但必须是整型。

例如数组定义：

```
int a[10];
```

表示定义了一个数组名为 a 的数组，一维，有 10 个元素，每个元素都是整型。

2、一维数组元素的引用形式为：

```
数组名[下标]
```

显然一个数组元素的引用代表一个数据，有时称这种形式的变量为下标变量，它和简单变量等同使用。

C 语言规定：数组元素的下标从 0 开始。在引用数组元素时要注意下标的取值范围。当定义数组元素的个数为 N 时．下标值取 0 到 N-1 之间的整数。例如上面定义的 a 数组共 10 个元素，下标值要在 0~9 之间。

下标可以是整型常量、整型变量或整型表达式。如：

```
    a[i]=a[9-i];
```

3、一维数组的初始化

对于程序每次运行时，数组元素的初始值是固定不变的场合，可在数组定义的同时，给出数组元素的初值。这种表达形式称为数组的初始化。C 语言规定只有静态数组和外部存储数组才能初始化。外部存储数组是在函数之外定义，往往在一个文件的开头定义，在整个文件范围内都能便用的数组。数组的初始化可用以下几种方法实现：

1）顺序列出数组全部元素的初值。

数组初始化时，将数组元素的初值依次写在一对花括号内。例如：

```
    int x[10]={0,-1,2,3,-4,5,6,-7,8,9};
```

2）只给数组的前面一部分元素设定初值。

例如：

```
    int y[10]={ 0,-1,2,3,-4};
```

定义数组 y 有 10 个元素，其中前 5 个元素设定了初值，分别为 0，-l，2，3，-4。而后六个元素未设定初值。C 系统约定，当一个数组的部分元素被设定初值后，对于元素为数值型的数组，那些未明确设定初值的元素自动被设定 0 值。所以数组 y 的后六个元素的初值为 0。但是，当定义数组时，如未对它指定初值，对于内部的局部数组，则它的元素的值是不确定的。

3）当对全部数组元素赋初值时，可以不指定数组元素的个数。

例如：

```
    int z[]={0,1,2};
```

系统根据花括号中数据的个数确定数组的元素个数。所以数组 z 有 10 个元素。但若提供的初值个数小于数组希望的元素个数时，则方括号中的数组元素个数不能省略。反之，如提供的初值个数超过了数组元素个数，就会引起程序错误。

例 0.21：用数组的形式求 10 个学生的平均成绩。

```
    #include <stdio.h>
    main()
    {
     int i,sum,score[10];
```

```
    float average;
    sum=0;
    for(i=0;i<10;i++)
    {
        scanf("%d",&score[i]);
        sum+=score[i];
    }
    average=(float)sum/10;
    printf("Average=%f\n",average);
}
```

在上面的程序中，数组 score 表示 10 个学生每个人的成绩。变量 sum 表示 10 个学生成绩的总和，而 i 是循环变量，用作数组的下标，浮点型变量 average 表示 10 人的成绩平均值。在 for 循环中，先输入每个学生的成绩，即给数组元素赋值。然后进行累加，求出总成绩。在循环结束后，得到总成绩，再求平均值。

0.8.2.2 二维数组

1、二维数组的定义

二维数组的定义形式为：

```
类型标识符 数组名[常量表达式][常量表达式];
```

其中各个部分的含义与一维数组中的各部分含义相同。

例如：

```
float a[2][3];
```

定义二维数组 a，它有 2 行 3 列。

由二维数组可以推广到多维数组。通常多维数组的定义形式有连续多个“[常量表达式]”。例如：

float b[2][3][4];

定义了三维数组 b。

C 语言把二维数组看作是一种特殊的一维数组，即它的成分又是一个数组。对于上述定义的数组 a，把它看作是具有 2 个元素的一维数组：a[0]和 a[l]，每个元素又是一个包含 3 个元素的一维数组。通常，一个 n 维数组可看作是一个一维数组，而它的元素是一个(n—1)维的数组。C 语言对多维数组的这种观点和处理方法，使数组的初始化、引

用数组的元素以及用指针表示数组带来很大的方便。

2、二维数组元素的引用

二维数组元素的引用形式为：

```
数组名[下标][下标]
```

通常 n 维数组元素的引用形式为数组名之后紧接连续 n 个“[下标]”。如同一维数组一样、下标可以是整型常量、变量或表达式。各维下标的下界都是 0。

3、二维数组的初始化

二维数组的初始化方法有以下几种：

1）按行给二维数组赋初值。

例如：

```
int x[2][3]={{1,2,3},{4,5,6}};
```

第一个花括号内的数据给第一行的元素赋初值，第二个花括号内的数据给第二行的元素赋初值，依次类推。这种赋初值方法比较直观。

2）按元素的排列顺序赋初值。

例如：

```
int x[2][3]={1,2,3,4,5,6};
```

这种赋初值方法结构性差，容易遗漏。

3）对部分元素赋初值。

例如：

```
int x[2][3]={{1,2},{3,4}};
```

其作用是使 x[0][0]=1，x[0][1]=2，x[1][0]=3，x[1][1]=4，其余元素均为 0。

4）如果对数组的全部元素都赋初值，定义数组时，第一维的元素个数可以不指定。

例如：

```
int x[][3]={1,2,3,,4,5,6};
```

系统会根据给出的初始数据个数和其他维的元素个数确定第一维的元索个数。所以数组 x 有 2 行。

也可以用分行赋初值方法，只对部分元素赋初值而省略第一维的元素个数，例如：

```
int x[][3]={{1,2},{}};
```

也能确定数组 x 共有 2 行。

例 0.22：说明一个二维数组 a，再对每一元素赋值，然后计算每一行各元素之和。

```
main()
{
 float a[2][3],b[2];              /*内部数组不能初始化*/
 int i;
 /*  给数组赋值*/
 a[0][0]=15.2; a[0][1]=10.2;a[0][2]=10.6;
 a[1][0]=6.7; a[1][1]=2.3; a[1][2]=10.3;
 for(i=0;i<2;i++)
 {
        b[i]=a[i][0]+a[i][1]+a[i][2];
        printf("%f %f %f %f\n",a[i][0],a[i][1],a[i][2],b[i]);
 }
}
```

程序开始定义了一个二维数组 a 和一个一维数组 b，数组 b 用来保存数组 a 每一行元素之和。然后给数组 a 各元素赋值，并计算各行元素之和，保存到数组 b 中，然后输出 a 的各行元素与每行元素的和。

程序运行结果为：

```
15.200000 10.200000 10.600000 39.000000
6.700000 2.300000 10.300000 22.299999
```

0.8.3 字符数组

字符数组是指每个元素都是字符型数据的数组，它的定义形式和元素的引用方法与一般数组相同。例如：

```
char a[20];
```

表示数组 a 有 20 个元素，每个元素都是字符型，能存放一个字符。

字符数组的初始化也与其他数组的初始化一样，对部分未明确指定初值的那些元素，系统自动设定 0 值字符即‘\0’，它是字符串结束标志符。

在 C 语言中，常用一维字符数组来存放一个字符串常量，习惯上把一维字符数组名称作字符串变量。由于字符数组在使用上的一些待殊性，故在此专门讨论。

0.8.0.1 字符数组与字符串

前面已经讨论过字符串，利用 C 语言关于字符串的规定，可以用字符串常量给字符数组初始化。例如：

```
char a[]={"C programe"};
```

也可以省略花括号，简单的写为：

```
char a[]="C programe";
```

注意，字符数组 a[]的元素个数是 11，不是 10。

需要指出，字符数组本身并不要求最后要有标志‘\0’。但当字符组需要作为字符串时，就必须有标志符‘\0’。例如：

```
char str1[]="teacher";
char str2[]={'t','e','a','c','h','e','r'};
```

则

```
printf("%s",str1);
```

是正确的，而

```
printf("%s",str2);
```

是错误的。后音将在输出 teacher 之后继续输出，直至遇到 8 位全 0 代码（即‘＼0’）为止。实际上字符数组 str1[]有 8 个元素；str2[]只有 7 个元素。

指定元素个数的字符数组用字符串常量给它初始化时，其元素个数不能少于字符串常量的字符数，但可等于。例如：

```
char str[]="abcde";
```

则 str[0]='a',str[1]='b',str[2]='c',str[3]='d',str[4]='e'。

0.8.0.2 字符数组的输入输出

字符数组的输入输出可以有两种方式：

1）逐个字符输入输出。用格式说明%c，结合循环控制结构输入输出一个字符。

2）将整个字符串一次输入输出。用格式说明%s，输入输出整个字符串。

例 0.23 输出字符串“string”。

```
main()
{
 char s[]="string";
 int i;
```

```
    for(i=0;s[i];i++)
        printf("%c",s[i]);
    printf("\n");
    printf("%s\n",s);
    }
```

程序开始定义了一个字符数组 s，然后分别用逐个字符输出和整个字符串一次输出的方法输出字符串。 输出结果都为：

```
string
```

需要注意的是：

1）输出的字符序列不包括字符串结束标志符“\0”。

2）用%s 格式输出字符串时，对应的输出项是字符串或字符数组名，而不是数组元素名。对于上例，写成

```
printf("%s\n",s[0]);
```

是错误的。因为数组元素 s[0]是字符，不是字符串。

3）字符串与存储字符串的字符数组的区别。字符串的有效字符是指从所指位置的一个字符开始至字符串结束标志符之前的那些字符。格式符“%s”只输出这些字符，而不会再继续输出字符串结束标志符之后的字符。例如：

```
char str="programming";
str[3]= '\0';
printf("%s\n"str);
```

将只输出：pro。

4）为字符数组输入字符串时，在 scanf 函数中输入项是字符数组名．不要加地址运算符‘&’。例如：

```
scanf("%s"str);
```

是正确的，而

```
scanf("%s"&str);
```

是不正确的。

5）用“%s”格式为字符数组输入字符串时，系统会自动在输入的有效字符列之后附加字符串结束标志符。如果用一个 scanf 函数输入多个字符串．则输入字符串以空格分隔。例如

```
char str1[6],str2[6],str3[6];
scanf("%s %s %s",str1,str2,str3);
```

输入数据：How are you?

则输入后，数组 str1，str2，str3 的内容分别是：

```
str1[0]='H',str1[1]='o',str1[2]='w',str1[3]='\0';
str2[0]='a',str2[1]='r',str2[2]='e',str2[3]='\0';
str3[0]='y',str3[1]='o',str3[2]='u',str2[3]='?',str2[4]='\0';
```

0.8.0.3 字符串处理函数

为了便于处理字符串，在 C 系统提供的函数库中包含有丰富的字符串处理函数，这里介绍其中几个比较常用的字符串处理函数。

1、字符串输出函数 puts

函数 puts 将字符串的内容输出到终端，并将字符串中的‘＼0’转换成换行符‘＼n’。即输出字符串内容，并换行。例如：

```
char str[]="1234";
puts(str);
```

将输出：

```
1234
```

2、字符串输入函数 gets(str)

其中，参数 str 是字符串，它的功能是从终端输入一行字符到 str 中。其中输入时的回车符被转换成‘\0’。str 不能是字符串常量。该函数调用将返回一个函数值，其值是 str 的起始地址。

3、求字符串长度函数 strlen(str)

其中，参数 str 是字符串，它的功能是返回 str 中的有效字符(不包括‘\0’)个数。例如：

```
char c[20]="12345";
printf("%d\n",strlen(c));
```

将输出 5。

4、字符串连接函数 stract(str1,str2)

其中，参数 str1、str2 是字符串，它的功能是将 str2 连接在 str1 的后面。str1 不能是字符串常量。函数调用返回一个函数值，函数值为 strl 的开始地址。正确使用该函数，要求 strl 必须足够大，以便能容纳 str2 的内容。注意，连接前，strl 和 str2 都各有自‘\0’。

连接后，strl 中的‘\0’在连接时被覆盖掉，而在新的字符串有效字符之后保留一个‘\0’。例如：

```
char str1[100]="Beijing",str2[]="China";
strcat(str1,str2);
puts(str1);
```

将输出 BeijingChina。

5、字符串拷贝函数 strcpy(str1, str2)和 strncpy(str1, str2)

其中，参数 str1、str2 是字符串。strcpy 函数的功能是将字符串 str2 拷贝到字符串 str1，限定 str1 不能是字符串常量，且 str1 定义得足够大，以便能容纳被拷贝的 str2 的内容。例如：

```
strcpy(str1,"computer");
```

strcpy 函数完成整个字符串的拷贝。在某些应用中，需要将一个字符串的前面一部分拷贝，其余部分不拷贝。调用函数 strncpy 可实现这个要求。strncpy(str1,str2,n) 的作用是将 str2 中的前 n 个字符拷贝到 strl(附加‘\0’)。其中 n 是整型表达式，指明欲拷贝的字符个数。如果 str2 中的字符个数不多于 n，则该函数调用等价于 strcpy(strl，str2)。

6、字符串比较函数 strcmp(str1, str2)

其中，参数 str1、str2 是字符串，它的功能是比较两个字符串大小。对两个字符串自左至右逐个字符相比较(按字符的 ASCII 代码值的大小)，直至出现不同的字符或遇到‘\0’为止。如全部字符都相同，则认为相等，函数返回 0 值；若出现不相同的字符，则以这第一个不相同的字符比较结果为准。若 strl 的那个不相同字符小于 str2 的相应字符，函数返回一个负整数；反之，返回一个正整数。

注意，对字符串不允许施加相等“==”和不相等“!=”运算，必须用字符串比较函数对字符串作比较。例如：

```
if(str1==str2) printf("Yes\n");
```

是非法的，而只能用

```
if(strcmp(str1,str2)==0) printf("Yes\n");
```

7、字符串字母转换函数

函数 strlwr(str)将字符串 str 中的大写字母转换成小写字母，而 strupr(str)则将字符串 str 中的小写字母转换成大写字母，其中 str 不能是字符串常量。

例 0.24：任意输入一个英文书名，将书名中的小写字母全部转换成大写字母，其余字符不变，输出转换后的书名。程序代码如下：

```
#include <stdio.h>
main()
{
 char str[100];
 int i;
 printf("Input the book name: ");
 gets(str);
 for(i=0; i<=strlen(str);i++)
        if(str[i]>='a'&&str[i]<='z')
              str[i]-=32;
 puts(str);
}
```

在上面的程序中，首先定义了一个字符数组 str，用来存储英文书名，然后用 gets 函数得到输入的英文书名。在 for 循环中，用 strlen 函数得到输入的英文书名字符串的长度，然后把英文书名的小写字母转换成大写字母。最后用 puts 函数输出转换后的英文书名。程序运行如下：

```
Input the book name: C programming
C PROGRAMMING
```

0.8.0.4 二维字符数组

二维字符数组用于处理多个字符串。定义方法类似二维整型数组和二维实型数组，例如：

```
char str[3][50];
```

定义了二维字符数组 str，str 中各下标变量的排列顺序为：

```
str[0][0],str[0][1],…,str[0][49]
str[1][0],str[1][1],…,str[1][49]
str[2][0],str[2][1],…,str[2][49]
```

在 C 语言中，可将二维数组看作是由若干个一维数组组成的数组。如上面的数组 str 可看作由三个一维数组组成的，这三个一维数组的数组名分别为 str[0]、str[1]和 str[2]。前面已说明，一维字符数组可用于处理一个字符串，所以数组 str 可用于处理三个字符

串。将三个字符串输入到数组 str 中的语句为：

```
for(i=0;i<3;i++)
 scanf("%s",s[i]);
```

或

```
for(i=0;i<3;i++)
 gets(s[i]);
```

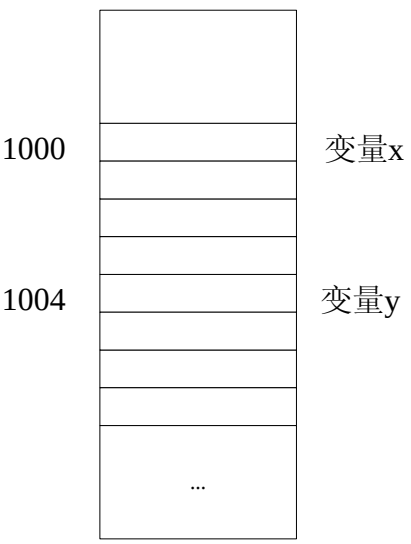
前者输入的串中不能包含空格，后者可包含空格。

0.9 指针

C 语言的设计者的初衷是设计一种能方便地、高效地实现描述系统软件的语言。指针正是为实现这一目标而精心设计的。在 C 语言中鼓励程序员灵活、正确地使用指针。利用指针可以简化程序，表达复杂的数据结构，灵活处理字符串，解决了调用函数返回多个值问题，为程序员直接参与处理内存地址提供了途径，尤其与数组那种灵活而变通的关系，大大提高了程序运行效率，节约了内存。

0.9.1 指针的概念

在程序中，如果定义了整型变量 int x，y；它意味着编译系统从某个内存地址开始为它们分配内存空间。如图 3—l 所示，系统为变量 x 分配了地址为 1000 的连续 4 个单元，为变量 y 分配了地址为 1004 的连续 4 个单元，而 x，y 这两个变量名在内存中并不存在。以后，在程序中对久 y 二变量的操作，变成了通过内存地址 1000 和 1004 对其内存单元的操作。例如程序中有 x＝6；和 y＝18；两个语句，编译系统实际上是将 6 送到地址为 1000 开始的 4 个连续单元中，将 18 送到地址为 1004 开始的 4 个连续单元中。1000 和 1004 为内存地址，而 6 和 18 分别为 1000 和 1004 单元地址中的内容。所以程序中一个变量包含了两个概念；



一个是变量在内存中的地址；一个是存放在该地址中的内容。由于这些工作由编译系统完成，用户是不需要关心的，用户只需在程序中直接引用变量。这种对变量的存取方式称为直接访问方式，也是为人们容易接受的方式。

由于通过内存地址可以访问到变量单元，称地址 图 3-1 为变量分配内存“指向”了该变量单元，所以一个变量的地址称为该变量的指针。简言之，指针即是内存地址。

0.9.2 指针变量

指针变量是用来存放内存地址的，因此指针变量就是存放指针的。如果定义了一个变量 p 用它来保存另一个变量 var 的地址，这样 p 就是指向 var 的指针变量。

0.9.2.1 指针变量的定义

指针变量即为变量，在使用它们之前需对其进行说明，其一般形式为：

```
类型标识符 *指针变量名;
```

其中，类型标识符表示指针变量指向的数据类型，符号*表示“指向…的指针”，指针变量名是用户自定义的标志符。如：

```
int *p;
float *q;
char* ch;
```

都是正确的指针变量定义。上述三个说明表示的意义是：

1）p 具有 int *类型，．即 p 是指向 int 类型的指针变量，以 4 字节为一个存取单位；

2）q 具有 f1oat *类型，即 q 是指向 float 类型的指针变量，以 4 个字节为一个存取单位；

3）ch 具有 char *类型，ch 是指向 char 类型的指针变量，以 1 个字节为一个存取单位。

0.9.2.2 指针变量的引用

C 语言中对指针变量的引用是通过两个运算符“＆”和“*”实现的。其中，&是取变量地址运算符，*是取指针变量所指向的变量的内容的运算符，也称为间接运算符。如：

```
int x,y;
```

```
int *p;
x=10;
p=&x;
y=*p;.
```

在变量说明部分 int *p；表示 p 是指向整型量的指针变量。p=＆x；通过取地址运算符＆将变量 x 的地址赋给了 p。再做 y=*p；这一赋值语句，表示把 p 所指向的变量的内容赋给 y，因此，执行完上述语句后 y=10。

0.9.2.3 指针变量的初始化

可以将一个指针变量初始化为给定数据类型的地址，或者为 0（NULL 值）。

在说明语句个对指针的类型作了限定，但是指针还没有与它所指向的具体对象建立联系，所以需要对它初始化。这项工作是由赋值表达式完成。

例如：

```
int x,*px;
px=&x;
```

就是完成对指针变量 px 的初始化工作，使 px 指向了变量 x，这时的 px 有了确切的值。

指针的初始化还可以在说明指针变量的同时置初值。如上述语句也允许写成下列形式：

```
int x,*px=&x;
```

但不能写成如下形式：

```
int *px=&x, x;
```

这是因为 x 的定义在后，先定义*p＝＆x 时 x 就是一个不定的量。

0.9.2.4 指针变量的运算

指针变量是用来存放内存地址的。内存地址是一种具有特定含义的整数，因此对指针变量只能进行有限的运算，概括起来有如下几个方面的运算。

1、算术运算

指针变量可以进行增量、减量运算；在一定的条件下进行减法运算；还可以与整形变量进行加减运算。

1）增量或减量运算

每当指针变量增量时，它将指向其基本类型的下一个元素的地址；每当指针变量减

量时，它将指向前一个元素的地址。例如在下面的程序代码中：

```
char *ps="Good morning!";
while(*ps)
putchar(*ps++);
```

在说明语句中，对指针变量 ps 赋初值，使它指向字符‘G’。字符串放在内存一片连续单元中，其中最后一个单元存放‘＼0’，标志字符串结束，其值为 0。在 while 循环的测试部分判断 ps 所指向的单元内容是否为 0。如果是 0，说明字符串结束，跳出循环；若不是 0，执行循环体 Putchar(*ps++)；把 ps 当前所指向的字符打印出来，同时指针变量 ps 加 1 指向下一个字符。注意，后缀++在这里是作用于 ps 而不是 ps 所指向的内容，原因就是“*”和“++”具有相同的运算优先级，且为左结合运算符。

2）两个指针变量在一定条件下可以做减法运算。

如果 p1 和 p2 两个指针变量指向同一数组时，则 p1-p2 的差值除以数据类型占用的字节数表示两指针变量之间相隔的数组元素的个数。

3）关系运算

指针变量在一定条件下可以进行比较。如果 p1 和 p2 都是同一类型指针变量，这两个指针变量之间可以进行<，>，<=，>=，==，!=操作。

若 p1<p2，则表示 p1 所指变量在 p2 所指变量之前。

若 p1==p2，则表示 p1 和 p2 都指向同一变量。

0.9.3 指针与数组

C 语言中指针与数组间有着很密切的关系。由于数组中的元素在内存中是连续排列存放，可以定义指针变量指向数组或数组元素，即把数组起始地址（称为数组的指针）或数组中某一元素的地址（称为数组元素的指针）存放到一个指针变量中。所以任何能由数组下标完成的操作都可由指针来完成。

一般来说，使用指针能使目标程序占用存储空间少，运行速度快。

下面的语句：

```
int a[3];
```

定义了一个包含 3 个元素的数组 a，它的元素 a[0]，a[1]，a[2]的地址分别表示为&a[0]，&a[1]，&a[2]。

在 C 语言中，数组名代表数组的起始地址。因此，数组名 a 和数组元素 a[0]的地址&a[0]是等价的。即数组的起始地址也就是数组中下标为 0 的元素的地址。由于在内存中数组元素的地址是连续的，所以通过数组 a 的起始地址加上一个数组元素所占用的空间就可依次求得每个元素的地址，假设 a[0]的地址（即 a 或 a+0）为 1000，则 a[1]的地址就可以用 a+1 表示，也就是说：a+1 指向数组 a 中下标为 1 的元素。它的实际地址为：1000+4（一个整型变量占 4 个字节）=1004。同样 a[i]的地址表示为 a+i，它的实际地址为 1000+i*4。对于初学者来说，1 经常会把 a[i]与 a+i 的含义混淆。a[i]表示数组 a 中第 i 个元素的值，而 a+i（即&a[i]）是 a[i]的地址。

通过上面的分析，可以很容易的知道，引用一个数组元素，既可以利用前面介绍的下标法（通过数组名和下标来实现），也可以利用地址法（通过给出的地址来访问）。例如，通过 a+1 地址可以找到数组元素 a[1]，即 a+1 地址所指向的元素。*(a+1)（地址法表示）与 a[1]（下标法表示）是等价的，都是数组 a 中下标为 1 的元素值。

在上节中，我们学习了可以定义一个指针变量指向一个变量，当然也可以定义一个指针变量指向数组元素。例如：

```
int *pa=&a[2];
```

这样，*p 就是 a[2]的值。

例 0.25：用指针变量指向数组元素，输出各元素的值。

```
#include <stdio.h>
main()
{
 int a[3],i=0;
 int *pa;
 a[0]=1;a[1]=2;a[2]=5;
 for(pa=a;pa<a+3;pa++)
 printf("a[%d]=%d\n",i++,*pa);
}
```

在上面的程序中，利用指针变量 pa 指向数组元素的方法输出各元素值。首先，pa

的初值等于 a，pa 指向数组中下标为 0 的元素 a[0]，*pa 就是 a[0]，在输出*pa 的值之后，pa++使指针 pa 指向下一个元素，此时 pa 指向 a[1]，*pa 就是 a[1]，在输出*pa 的值之后，pa++又使指针 pa 指向 pa 指向下一个元素，…，直到 pa=a+3 为止，从而输出完 a 数组的三个元素值。

在使用指针变量时，要注意以下几个问题：

1）指针变量 pa 的数值可以不断变化，因此 pa 可以指向不同数组元素。

2）数组名 a 代表数组的起始地址，它是一个常数，不能做自加（或自减）运算，因此它不能改变其指向。

3）在使用指针法访问数组元素时，注意“下标是否越界”的问题。例如上例中定义的 a 数组，它只有 3 个元素 a[0]，a[1]，a[2]。如果在程序中引用 a[3]或其他值，C 编译系统并不对其指出“下标越界”错误，从而使程序运行后可能得到意料之外的结果。

0.9.4 指针与函数

指针既可以作为函数间传递的参数，也可以作为函数的返回值类型。本节主要介绍函数与指针的关系以及如何将指针与函数配合使用。

0.9.4.1 指针变量作为函数参数

在 C 语言中，函数的参数传递有 2 种方式：传递值和传递地址。前面讲过的整型数据、实型数据或字符型数据等都可以作为函数参数进行传递，这些类型数据传递的是变量的值，那么就称它为“传递值”方式。在学习了指针变量的概念后，就可以进一步学习用指针变量作为函数参数。指针变量的值是一个地址，指针变量作函数参数时，传递的也是一个变量的值，但这个值是另一个变量的地址，这种把 变量的地址传递给被调用函数的方式称为“传递地址”方式。

例 0.26：利用指向一维数组元素的指针变量作为函数参数，将数组中各元素按相反顺序输出。

```
void swap(int *px, int *py)
{
```

```
 int x;
 x=*px;
 *px=*py;
 *py=x;
}

main()
{
 int i,a[10]={0,1,2,3,4,-5,6,-7,8,-9};
 int *p1, *p2;
 printf("The original array:\n");
 for(i=0;i<10;i++)
        printf("%d   ",a[i]);
 printf("\n");
 for(i=0;i<5;i++)
 {
        p1=&a[i];
        p2=&a[10-1-i];
        swap(p1,p2);
 }
 printf("The inverted array:\n");
 for(i=0;i<10;i++)
        printf("%d   ",a[i]);
 printf("\n");
}
```

运行结果如下：

```
The original array:
0   1   2   3   4   -5   6   -7   8   -9
The inverted array:
-9   8   -7   6   -5   4   3   2   1   0
```

函数 swap 的作用是交换 px 和 py 所指向的 2 个变量的值。由于在主函数中 p1 和 p2 指向的是 2 个数组元素，也就是说 p1 和 p2 是 2 个指向数组元素的指针变量作为实参。swap 函数被调用时，p1 和 p2 传递给了 px 和 py，并在函数中完成了 px 和 py 所指向的变量的值的交换，也就是交换了 2 个数组元素的值。

0.9.4.2 函数的返回值为指针变量

函数的返回值可以是整型、实型、字符型等一些基本类型，但也可以是一个指针，它可以是指向整型、实型、字符型以及结构等类型的指针。

这种返回指针值的函数，一般定义形式为：

```
类型标识符   *函数名(形式参数表)
{函数体}
```

其中，类型标识符表示了函数返回的指针变量类型。*代表函数的返回值是一个指针，其他部分的含义与一般函数定义相同。

例 0.27：返回值为指针变量的函数。

```
main()
{
char *strcon(char ch);
char *ps, s='A';
ps=strcon(s);
printf("%s \n",ps);
}

char *strcon(char ch)
{
static char sa[10];
int i;
for(i=0;i<6; i++)
sa[i]=ch-i;
sa[6]='\0';
return(sa);
}
```

运行结果：

```
A@?>=<
```

在上例中，将字符'A'传递给函数 strcon，在 strcon 中将 ASCII 码小于'A'的字符依次存放到字符数组 sa 中，最后返回字符数组 sa 的首地址给主函数中的指针变量 sp。

0.9.4.3 指向函数的指针

指针就是内存空间的地址。函数是一段代码，函数在运行时也是存放在内存中的，也需要占有内存的存储单元。函数指针就是函数在内存中的首地址，也就是函数运行的入口地址。与变量相似，函数名就代表函数的入口地址。函数名如果单独用在表达式中，就是一个函数指针常量。

函数指针变量就是一个存储函数指针的指针变量。在函数指针变量定义中，必须说明指向的函数的返问类型和所需参数，函数指针变量的定义形式与函数说明的形式比较相似，如下所示：

```
类型标识符 (*指针变量名)(形式参数表);
```

函数指针变量可以象函数名一样进行函数调用。所不同的是，函数名是一个函数指针常量，而函数指针变量则可以在程序运行过程中通过赋值指向不同的函数（但函数的返回类型、参数个数和参数类型应当是相同的)，在程序运行过程中来决定要调用的函数。

例 0.28：通过函数指针调用函数。

```
#include <stdio.h>
int sum(int x, int y)
{
 return (x+y);
}
main()
{
 int a=5, b=3, c,d;
 int (*pfunc)();
 pfunc=sum;
 c=(*pfunc)(a,b);
 d=sum(a,b);
 printf("%d+%d=%d\n",a,b,c);
 printf("%d+%d=%d\n",a,b,d);
}
```

运行结果如下：

```
5+3=8
5+3=8
```

说明：

1）函数 sum 是返回值为整型的函数，pfunc 是指向一个指向函数（此函数的返回值为整型）的指针变量。注意使用圆括号把*pfunc 括起来，否则 C 编译系统将把它视为返回值为指针（该指针指向整型变量）的函数。

2）赋值语句“pfunc=sum;”的作用是：将函数的入口地址赋给指针变量 pfunc。如同数组名代表数组起始地址一样，函数名代表函数的入口地址，无需任何括号和参数，更不需要使用地址运算符“&”。这时 pfunc 就是指向函数 sum 的指针变量。

3）赋值语句“c=(*pfunc)(a,b);”中的（*pfunc）就是调用函数 sum，即通过指向函数的指针变量 pfunc 间接调用函数 sum，（*pfunc）后面的实参表必须与 sum 中定义的形参表一致。调用后的返回值是一整型变量值，即 x+y 的值。

4）程序中分别用函数名和指向函数的指针变量来调用 sum，从运行结果来看，2 种调用方法是相同的。

5）指向函数的指针变量只能指向函数的入口处，而不能指向函数中的某一条具体语句，因此像 pfunc++，pfunc--，pfunc+n 等这样的指针运算是没有意义的。

通过函数指针变量，可以使程序具有更大的灵活性。

0.9.5 多级指针

一个指针可以指向任何一种数据类型，包括指向一个指针。当指针变量 p 中存放另一个指针 q 的地址时，则称 p 为指针型指针，也称多级指针。本节介绍二级指针的定义及应用。

指针型指针的定义形式为：

```
类型标识符  ** 指针变量名
```

由于指针变量的类型是被指针所指的变量的类型，因此，上述定义中的类型标识符应为被指针型指针所指的指针变量所指的那个变量的类型。

为指针型指针初始化的方式是用指针的地址为其赋值，例如：

```
int x;
int *px;
int **q;  /* 定义二级指针 q */
```

若有：

```
px=&x;
```

则在程序中，使用*px 等价与使用 x，成为对 x 的间接访问。

对二级指针若有：

```
q=&px;
```

则使用*q 即间接访问二级指针等价于使用 px，再次间接访问二级指针：

```
**q=*(*q)=*px=x;
```

因此，对一个变量 x，在 C 语言中，可以通过变量名对其进行直接访问，也可以通过变量的指针对其进行间接访问（一级间接），还可以通过指针型指针对其进行多级间接访问。

例 0.29：使用二级指针引用字符串。

```
#define   LENGTH 6
#include <stdio.h>
main()
{
 char *pstr[]={"Beijing", "NewYork", "Paris", "London", "Washington", "Tokyo"};
 char **p;
 int i;
 for(i=0;i<LENGTH; i++)
 {
        p=pstr+i;
        printf("%s \n", *p);
 }
}
```

在上面程序中，p 是指针型指针，在循环开始 i 的初值为 0，语句 p=pstr+i;用指针“数组 pstr 中的元素 pstr[0]为其初始化，*p 是 pstr[0]的值，即字符串"Beijing"的首地址，调用函数 printf 以％s 形式就可以输出 pstr[0]所指字符串。pstr+i 将指针向后移动，依次输出其余各字符串。程序运行结果为：

```
Beijing
NewYork
Paris
London
Washington
Tokyo
```

0.10 结构类型及其它构造类型

在实际生活中，有着大量由不同性质的数据构成的实体，如学生的学籍登记表由姓名、学号、性别、年龄、学习成绩五个类型不同的数据项组成。其中，姓名、性别用字符串表示；学号、年龄可用整数表示；学习成绩可用浮点数表示对于这样的数据形式，数组将无法精确描述。因此，在 C 语言中提供了一种新的称为结构的构造型数据类型。结构是一组相关的不同类型的数据的集合。结构类型为处理复杂的数据提供了便利的手段。本节讨论结构的定义、说明和使用，结构与数组和指计等基本问题，并介绍联合的基本概念，以及怎样用 typedef 定义新的类型。

0.10.1 结构类型

结构与数组类似，都是由若干分量组成的。数组是由相同类型的数组元素组成，但结构的分量可以是不同类型的，结构中的分量称为结构的成员。访问数组中的分量(元素)是通过数组的下标，而访问结构中的成员是通过成员的名字。

0.10.1.1 结构类型的概念与定义

在程序中使用结构之前，首先要对结构的组成进行描述，称为结构的定义。结构的定义说明了该结构的组成成员，以及每个成员的数据类型。结构定义的一般形式如下：

```
struct  结构类型名称
{       数据类型  成员名 1;
        数据类型  成员名 2;
        …
        数据类型  成员名 n;
};
```

其中：struct 为关键字，是结构的标识符；结构类型名称是所定义的结构的类型标识，由用户自己定义；{}中包围的是组成该结构的成员项；每个成员的数据类型既可以是简单的数据类型，也可以是复杂的数据类型。整个定义作为一个完整的语句用分号结束。

结构类型名称是可以省略的，此时定义的结构称为无名结构。

虽然结构定义时并不要求其内部成员之间有任何内在的联系，但一般来说，结构中所有的成员在逻辑上都是彼此紧密相关的，将毫无任何逻辑关系的一组成员放入同一结构中没有任何实际意义。

为了描述学生登记表，可以定义如下结构：

```
struct student{
 char  name[20]; /*姓名，字符数组作为结构成员 */
 int    no;          /*学号，整型数据作为结构成员 */
 char  gender[6]; /*性别，字符数组作为结构成员 */
 int    age;         /*年龄，整型数据作为结构成员 */
 float score;              /*成绩，浮点型数据作为结构成员 */
};
```

在上述结构定义中，结构类型名称为 student，可以称这个结构类型为 student。在 student 中，有 5 个成员 name，no，gender，age，score，它们分别为字符数组，整型数

据，字符数组，整型数据和浮点型数据。

结构的定义明确了结构的组成形式，定义了一种 C 语言中原来没有、而用户实际需要的新的数据类型。与其他的数据类型不同，在程序编译的时候结构的定义并不会使系统为该结构分配内存空间，只有在说明结构变量时才分配内存空间。

在程序中，结构的定义可以在一个函数的内部，也可以在所有函数的外部，在函数内部定义的结构，仅在该函数内部有效，而定义在外部的结构，在所有函数中都可以使用。

0.10.1.2 结构变量的说明

结构的定义说明了它的组成，即变量在计算机中的存在格式，要使用该结构就必须说明结构类型的变量。结构变量说明的一般形式如下：

```
struct 结构类型名称 结构变量名;
```

定义结构便是定义了一种由成员组成的复合类型，而用这种类型说明了一个变量，才会产生具体的实体。说明结构变量的作用类似于说明一个 int 型的变量一样，系统为所说明的结构变量按照结构定义时的组成，分配存储数据的实际内存单元。结构变量的成员在内存中占用连续存储区域，所占内存大小为结构中每个成员的长度之和。

如：将变量 var 说明为 student 型的结构变量：

```
struct student var;
```

也可以说明多个 student 型的结构变量：

```
struct student var1, var2, var3;
```

一个结构变量占用内存的实际大小，可以使用 sizeof 运算求出。sizeof 是单目运算，其功能是求出运算对象所占的内存空间的字节数目。使用的一般形式为：

sizeof（变量或类型说明符）

例 0.30：sizeof 运算。

```
#include "stdio.h"
main()
{
 char str[20];
 struct student{          /*定义结构 student */
```

```
    char  name[20];
    int           no;
    char  gender[6];
    int    age;
    float  score;
    } var;                    /*说明结构变量 var */
    printf("char: %d \t",sizeof(char));       /* char 型的长度  */
    printf("int:   %d\t",sizeof(int));        /*int  型的长度  */
    printf("long: %d\t",sizeof(long));        /*long 型的长度  */
    printf("float: %d\t",sizeof(float));      /*float 型的长度  */
    printf("double: %d\t",sizeof(double));    /*double 型的长度  */
    printf("str: %d\t",sizeof(str));                /*变量 str 的长度  */
    printf("student: %d\t",sizeof(struct student)); /*结构 student 的长度  */
    printf("var: %d\t\n",sizeof(var));              /*结构变量 var 的长度*/
    }
```

程序运行结果为：

```
    char: 1          int:   4 long: 4 float: 4          double: 8          str: 20 student: 40          var:
40
```

0.10.1.3 结构成员的引用

结构作为若干成员的集合是一个整体，但在使用结构时，不仅要对结构整体进行操作．而且更频繁的是要访问结构中的每个成员。在程序中使用结构中成员的方法为：

```
        结构变量名.成员名称
```

如要将“王小明 080314 男 15 95.5”这个学生信息输入到 var 这个结构变量，只能对其各个成员分别赋值：

```
    strcpy(var.name,"王小明"); /*为结构中的字符串成员赋值,注意，不能写成 var.name=" 王小明
"*/
    var.no=0314;              /*为结构中的整型成员赋值*/
    strcpy(var.gender,"男");          /*为结构中的字符串成员赋值*/
    var.age=15;                       /*为结构中的整型成员赋值*/
    var.score=95.5;           /*为结构中的浮点型成员赋值*/
```

在 C 语言中，指明结构成员的符号“.”是一种运算符，它的含义是访问结构中的成员。这样 var.no 的含义就是访问结构变量 var 中的名为 no 的成员。“.”操作的优先级在 C 语言中是最高的，其结合性为从左到右。

在 C 语言中，结构的成员可以象一般变量一样参与各种操作和运算，而作为代表结构整体的结构变量，要进行整体操作就有很多限制，由于结构中各个成员的逻辑意义不

同，类型不同，对结构变量整体的操作的物理意义不是十分明显。C 语言中能够对结构进行整体操作的运算不多，只有赋值“=”和取地址“&”操作。例如：

```
struct student var1,var2;
 var1=var2;
```

进行结构变量整体赋值，即完成对应分量的赋值，就是将结构变量 var2 的值按照各个分量的对应关系赋给结构变量 var1。这里“=”两侧的结构变量类型必须是相同的结构类型。

0.10.1.4 结构变量的初始化

在结构说明的同时，可以对每个成员置初值，称为结构的初始化。结构初始化的一般形式如下：

```
struct  结构类型名称   结构变量={初始化数据};
```

其中”{}”包围的初始化数据用逗号分隔。初始化数据的个数与结构成员的个数应相同，它们是按成员的先后顺序一一对应赋值的。此外，每个初始化数据必须符合与其对应的成员的数据类型。例如：在前面给出的 student 类型的变量，可以用如下形式进行初始化。

```
struct student var={"王小明",0314,"男",15,95.5};
```

与数组的初始化特性相同，结构变量的初始化仅限于对外部的和静态的结构变量，对于存储类型为 auto 型的结构变量，不能在函数内部进行初始化，只能使用赋值语句进行赋值。

0.10.2 结构数组

结构数组是一个数组，其数组中的每一个基本元素都是结构类型。说明结构数组的方法是：先定义一个结构，然后用结构类型说明一个数组变量。

例如：为记录 100 个学生的基本情况。可以说明一个有 100 个元素的数组。每个元素的基类型为一个结构，在说明数组时可以写成：

```
struct student stu[100];
```

stu 就是有 100 个元素的结构数组，数组的每个元素为 student 型结构。

要访问结构数组中的具体结构，必须遵守数组使用的规定，按数组名及其下标进行访问，要访问结构数组中某个具体结构下的成员，又要遵守有关访问结构成员的规定，使用“.”访问运算符和成员名。访问结构数组成员的一船格式是：

```
结构数组名[下标].成员名
```

同一般的数组一样，结构数组中每个元素的起始下标从 0 开始，数组名称表示该结构数组的存储首地址。结构数组存放在一连续的内存区域中，它所占内存数目为结构类型的大小乘以数组元素的个数。

例如，将数组 stu 中的 2 号元素赋值为：“王小明”，0314，“男”，15，95.5，就可以使用下列语句：

```
strcpy(stu[2].name,"王小明"); /*为结构中的字符串成员赋值,注意，不能写成 var.name=" 王小明"*/
stu[2].no=0314;                /*为结构中的整型成员赋值*/
strcpy(stu[2].gender,"男");        /*为结构中的字符串成员赋值*/
stu[2].age=15;                 /*为结构中的整型成员赋值*/
stu[2].score=95.5;             /*为结构中的浮点型成员赋值*/
```

0.10.3 结构指针

结构指针是指向一种结构类型的指针变量，它是结构在内存中的首地址。结构指针说明的一般形式是：

```
struct 结构类型名称 * 结构指针变量名;
```

例如：

```
struct student *pstu1, stu2;
pstu1=&stu2;
```

说明了两个变量，一个是指向结构 student 的结构指针 pstu1，stu2 是一个 student 结构变量。pstu1=&stu2 是 pstu1 指向了结构变量 stu2。

通过结构变量 stu2 访问其成员的操作，也可以用等价的指针形式表示：

```
stu2.no=0314;
```

等价于

```
(*pstu1).no=0314;
```

由于运算符“*”的优先级比运算符“.”的优先级低，所以必须有（）将*pstu1 括起来。若省去括号，则含义就变成了*(pstu1.no)。

在 C 语言中，通过结构指针访问成员可以采用运算符“->”进行操作，对于指向结构的指针，为了访问其成员可以采用下列语句形式：

结构指针->成员名；

这样，上面通过结构指针 pstu1 访问成员 no 的操作就可以写成：

```
pstu1->no=0314;
```

如果结构指针 p 指向一个结构数组，那么对指针 p 的操作就等价于对数组下标的操作。

结构指针具有一般指针的特性，如在一定条件下两个指针可以进行比较，也可以与整数进行加减。但在指针操作时应注意：进行地址运算时的放大因子由所指向的结构的实际大小决定。

例 0.31：结构数组和结构指针的使用。

```
#include <stdio.h>
main()
{
 int i;
 struct date{                          /* 定义结构 date */
 int    year;
 int    month;
 int day;
 };
 struct date *pdate, days[2];   /*定义结构指针变量 pdate 和结构数组 days */
 days[0].year=2008;
 days[0].month=2;
 days[0].day=15;
 days[1].year=2007;
 days[1].month=9;
 days[1].day=14;
 for(i=0;i<2;i++)
 {
        pdate=&days[i];
        printf("year %d ,month %d, day %d\n", pdate->year,pdate->month,pdate->day);
 }
}
```

在上面的程序中，首先定义了一个结构变量 date，包含 3 个整型数据成员：year，month，day，然后定义了 date 结构指针变量 pdate 和结构数组 days，并分别给 days 元素

中的各成员赋值。程序最后通过循环，让结构指针变量 pdate 依次指向 days 中的各元素，并输出 days 中各元素的成员。

程序运行结果为：

```
year 2008 ,month 2, day 15
year 2007 ,month 9, day 14
```

0.10.4 结构与函数

结构在函数方面的应用表现在如下两个方面：（1）结构变量和指向结构变量的指针可以作为函数参数。（2）结构变量和指向结构变量的指针可以作为函数的返回值。。

0.10.4.1 结构变量和指向结构变量的指针作为函数参数

1、结构变量作为函数参数

结构变量作函数参数时，形参和实参都要求是同一种结构模式的结构变量。函数调用时，系统将实参拷贝一个副本给形参，这种调用方式在被调用函数中无法改变调用函数的参数。

例 0.32：结构变量作为函数参数。

```
#include <stdio.h>
struct complex{
 float real,image;
};
main()
{
 static struct complex x={1.0,2.0},y={0.0,4.0};
 struct complex z1,z2,add(),multiply();
 z1=add(x,y);
 z2=multiply(x,y);
 printf("(%f+i%f)+(%f+i%f)=",x.real,x.image,y.real,y.image);
 printf("%f+i%f\n",z1.real,z1.image);
 printf("(%f+i%f)*(%f+i%f)=",x.real,x.image,y.real,y.image);
 printf("%f+i%f\n",z2.real,z2.image);
}
struct complex add(struct complex x,struct complex y)
{
 struct complex z;
 z.real=x.real+y.real;
 z.image=x.image+y.image;
```

```
 return z;
}
struct complex multiply(struct complex x,struct complex y)
{
 struct complex z;
 z.real=x.real*y.real-x.image*y.image;
 z.image=x.real*y.image+x.image*y.real;
 return z;
}
```

在上面的程序中，定义了结构 complex，代表一个复数，其中，real 用来存放复数的实部，image 用来存放复数的虚部。程序由 1 个 main 函数和 2 个被调用函数 add 和 multiply 组成，它们分别完成 2 个复数的加法和乘法运算。2 个被调用函数的形参都是 complex 型的结构变量，结构变量可作为函数的参数。另外，这两个函数的返回值又都是 complex 型的结构变量，结构变量可作为函数的返回值，这种函数称为结构函数。

2、指向结构变量的指针作为函数参数

结构变量可以作为函数的参数，前面举过的例子中已经出现。指向结构变量的指针作函数的形参，要求对应的调用函数的实参用相同类型的结构变量的地址值，实现传址调用，在被调用函数中可通过改变形参指针所指向结构变量的值来改变实参的值。另外，这种传址调用比用结构变量作函数参数实现的传值调用具有更高的效率。因为传值调用时，系统将拷贝实参的副本给被调用函数的形参。而传址调用时，只传递其地址值．特别是当结构变量比较复杂时，传址调用要花费较多的时间和占用较大的空间，因此效率较低。因此，在实际应用中较多地使用指向结构变量的指针作函数参数。

例 0.33：以指向结构变量的指针作为函数参数，仍以前面的程序为例。

```
#include <stdio.h>
struct complex{
 float real,image;
};
main()
{
 static struct complex x={1.0,2.0},y={0.0,4.0};
 struct complex z1,z2,add(),multiply();
 z1=add(&x,&y);
 z2=multiply(&x,&y);
```

```
 printf("(%f+i%f)+(%f+i%f)=",x.real,x.image,y.real,y.image);
 printf("%f+i%f\n",z1.real,z1.image);
 printf("(%f+i%f)*(%f+i%f)=",x.real,x.image,y.real,y.image);
 printf("%f+i%f\n",z2.real,z2.image);
}
struct complex add(struct complex *px,struct complex *py)
{
 struct complex z;
 z.real=px->real+py->real;
 z.image=px->image+py->image;
 return z;
}
struct complex multiply(struct complex *px,struct complex *py)
{
 struct complex z;
 z.real=px->real*py->real-px->image*py->image;
 z.image=px->real*py->image+px->image*py->real;
 return z;
}
```

该程序中，在被调用函数 add 和 multiply 中都使用了指向结构变量的指针作函数形参，调用时所对应的实参为相同结构类型的结构变量的地址值。在这两个函数中，使用指向结构变量的指针都不是为了改变调用函数中的实参值，而是为提高函数调用时传递数据的效率。

0.10.4.2 结构变量和指向结构变量的指针作为函数返回值

结构变量和指向结构变量的指针都可以用作函数的返回值。结构变量作函数返回值，该函数称为结构函数。

1、结构函数

在前面讲过的例子中，出现了两个结构函数 add 和 multiply。这两个函数的返回值都是结构名为 complex 的结构变量。

结构函数在应用中也是常出现的，由于前面已举过例子，这里不再详述。

2、指向结构变量的指针作为函数的返回值

函数的返回值可以是结构变量，也可以是指向结构变量的指针。

例 0.34：根据学生的学号查找学生成绩。

```
sruct student{
 int no;
 float score;
} stu[4]={200801,93,200802,95,200803,92.5,200804,97};
main()
{
 struct student *ps, *find();
 int i,no[3];
 no[0]=200803; no[1]=200806;no[2]=200802;
 for(i=0;i<3;i++)
 {
        ps=find(no[i]);
        if(ps!=0)
        printf("No.: %d, Score: %f\n",ps->no,ps->score);
        else
        printf("Error No.!\n");
 }
}

struct student *find(int no)
{
 int i;
 for(i=0;i<4;i++)
 {
        if(stu[i].no==no)
        return &stu[i];
 }
 return 0;
}
```

在上面的程序中，定义了 student 结构，其中，no 用来保存学生的学号，score 用来保存学生的成绩。然后定义了 student 结构数组 stu[4]。函数 find 根据输入的学号，在结构数组中进行查找，如果找到对应的学号，返回指向该结构变量的指针。如果输入的学号不在结构数组中，则返回的指针值为 0。它所返回的地址值在主函数中赋给结构变量指针 ps。然后程序根据 ps 的值输出相应的信息。

程序运行结果如下：

```
No.: 200803, Score: 92.500000
Error No.!
No.: 200802, Score: 95.000000
```

0.10.5 位段

位段是一种对信息进行的压缩的方法，信息经压缩后可以节省内存空间。

0.10.5.1 位段的概念

位段是一种压缩信息的方法，它所使用的是一种结构的数据形式。该方法是在结构中定义一种特殊的成员，该成员是以位为单位定义长度的。采用这种压缩信息方法的好处在于对各数据值的存取可采用结构成员的存取方法，操作起来十分方便。

在一个结构中可以有若干个被定义的位段式的成员，其格式如下：

```
unsigned 成员名: 二进制位数
```

位段式成员一般为 unsigned int 型的，成员名同标识符，即与一般结构成员名命名法相同。冒号后面写有该位段的长度，以二进制位为单位。例如：

```
struct data_struct{
        unsigned    var1:1;
        unsigned    var2:1;
        unsigned    var3:1;
        unsigned    var4:3;
        unsigned    var5:5;
        int         var6;
}
```

这是一个具有 6 个成员的结构模式，结构名为 data_struct。前三个成员都是只占 1 位内存空间，接着 2 个成员分别占 3 和 5 位内存空间，最后一个成员不是位段形式，它是一个 int 型变量。

下面定义一个结构变量 ds，其格式如下：

```
struct data_struct ds;
```

按结构成员的方法对其每个位段式成员进行操作。例如，将 6 赋给 var4，将 8 赋给 var5 可用如下方式：

```
ds.var4=6;
ds.var5=8;
```

可见，对位段式变量进行操作十分方便。

这里，需要注意的是位段所允许的最大致值范围。例如，下面操作：

```
dl.var4=8;
```

是错误的。因为成员 var4 的长度是 3 个二进制位，它的最大取值为二进制数 111，即十进制数 7。如果树 8 赋给 ds.var4，则 var4 只能取后面 3 位，即 000 了。

0.10.5.2 使用位段时应注意的事项

1）在位段结构中可以定义无名位段，它可以用来作为位段的分隔。例如：

```
struct bitsegment{
        unsigned    type:4;
        unsigned:4;
        unsigned    count:8;
}
```

该结构中有三个位段成员，其中有一个无名位段，它占 4 位，它的作用是将 type 成员和 count 成员用 4 个空位分隔开，即在 type 成员后面空 4 位不用，接着再是 count 成员占 8 位。

2）长度为 0 的位段用来使字边界对齐。使用长度为 0 的位段可使下一个位段从一个新的字开始存放。例如：

```
struct a{
        unsigned    x:2;
        unsigned    y:8;
        unsigned    : 0;
        unsigned    z:4;
        unsigned    w:8;
}
```

该结构中有 5 个位段成员，其中有一个长度为 0 的无名位段，它使得 x 和 y 成员存放在一个字中，但是没有占满，而 z 和 w 成员存放在另一个字中。

3）一个位段不能跨越两个字，只能存放在同一个字中。如果前面的几个位段几乎占满一个字，空下来的不足以再放一个位段时，该位段只好从下一个字开始。

4）不能构造位段数组，也不能对位段变量进行地址操作。

5）位段可以用整型格式输出，即可用%d，%o 和%x 等格式符进行输出。位段可在表达式中引用，系统自动将位段转换成 int 型数。

6）位段不能定义在联合中，位段也不能作为函数的返回值。

0.10.6 联合类型

联合是一种由不同类型数据组成的复合数据类型。 在一个联合内可以定义多种不同的数据类型，在一个被说明为该联合类型的变量中，先许装入该联合所定义的任何一种数据。这在前面的各种数据类型中都是办不到的。例如，定义为整型的变量只能装入整型数据，定义为实型的变量只能赋予实型数据。

联合与结构有一些相似之处，也有本质上的不同。第一点不同在于存储空间的分配。在结构中各成员有各自的内存空间，一个结构变量的总长度是各成员长度之和。而在联合中，各成员共享一段内存空间，一个联合变量的长度等于所有成员中最长成员的长度。可以使用以下通俗地比喻：结构好比每人都有自己的衣服，并且为“量体裁衣、度身定做”；而联合则是按照个子最大的人作衣服，大家都可穿这个“大”衣服。这样，联合可以覆盖地使用内存，极大地提高了内存的使用效率。总之，联合对内存的使用可以说是“取长补短”。

第二点不同在于变量的使用。联合变量可被赋予任一成员值，但每次只能赋一种值，赋入新值则冲去旧值。但是结构的变量则不可这样使用。正是这样特点，使联合变量比结构变量灵活。

联合类型的定义和联合变量说明的格式与结构非常相似，只是把 struct 改为 union 而已。

0.10.6.1 联合的定义

联合的定义形式如下：

```
union 联合名
{
        数据类型 成员名 1;
        数据类型 成员名 2;
        …
        数据类型 成员名 n;
}
```

其中，union 为关键字，是联合的标识符；联合名称所定义的联合的类型标识，由用户自己定义。其他部分的含义与结构对应部分的含义相同。例：

```
union union_a{
        char  x;
        short y;
        long  z;
}
```

这段程序定义了一个名为 union_a 的联合类型。它含有 3 个成员，一个为字符型，成员名为 x；另一个为短数值型，名为 y；最后一个为长数值型，名为 z。系统在进行内存分配时，按照 long 型为联合 a 分配内存。

0.10.6.2 联合变量的说明

一个联合类型必须经过定义之后，才能把变量说明为该联合类型。

联合变量的说明和结构变量的说明方式相同，联合变量说明的一般形式如下：

```
union  联合类型名称  联合变量名;
```

如：将变量 var 说明为 union_a 型的联合变量：

```
union union_a    var;
```

也可以说明多个 union_a 型的联合变量：

```
union union_a    var1, var2, var3;
```

0.10.6.3 联合变量的赋值和使用

对联合变量的赋值、使用都只能是对变量的成员进行。联合变量的成员表示为：

```
联合变量名.成员名
```

例如：var 被说明为 union_a 类型的变量之后，可使用：

```
var.x=5;
var.y=3;
```

不允许只用联合变量名作赋值或其它操作。也不允许对联合变量作初始化赋值，赋值只能在程序中进行。

还要再强调说明的是，一个联合变量每次只能赋予一个成员值。换句话说，一个联合变量的值就是联合变量的某一个成员值。

例 0.35：联合的使用。

```
union    un_a{
 int           i;
 float  f;
 double       d;
```

```
 long   int   li;
} x;
main()
{
 x.i=6;
 printf("%d\n",x.i);
 x.d=3576266;
 printf("%f\n",x.d);
 x.f=2340.3434;
 printf("%f \n",x.f);
 x.li=3423499;
 printf("%d \n",x.li);
}
```

在上面的例子中，定义了联合类型 un_a，并说明了一个 un_a 类型的变量 x，然后分别给 x 中的各个成员分别赋值，然后输出该成员值。程序运行结果如下：

```
6
3576266.000000
2340.343400
3423499
```

使用联合应注意的问题是：

1）内存覆盖

尽管在联合所使用的内存中可以存储不同的成员，即可以存储不同类型的数据，但每一刻只能存储一个成员。通俗地说，只有一套衣服，一次只能由一个演员来穿。

2）以新替旧

联合变量存储的是最后一次存储的成员的内容。换言之，变量存储了新成员，则自动替换了老成员。

3）地址唯一

联合变量和各个成员拥有同一地址。

4）变量不可赋值

不能给联合的变量名赋值，也不能对联合的变量名初始化及引用联合的变量名，使用的都是联合变量的成员。

5）嵌套

联合与结构可以互相嵌套，数组也可以作为联合的成员，也可以定义联合数组。

6）不能用作函数

不能把联合变量作为函数的参数，也不能使函数带回联合变量，但是，可以使用联合变量的指针。

0.10.7 枚举类型

为了增加程序代码的可读性，C 语言允许使用枚举类型数据。

当一个变量的值只有有限的几种可能情况时，就可以引用枚举类型了。那么，什么是枚举呢？一般认为，枚举是一种机制，用于定义一组已命名常量。也可以说，枚举是将变量的值一一列举出来．并且变量的值只限于列举出来的范围内。

枚举类型的一般定义形式：

```
enum 枚举类型 {枚举常量名表};
```

其中 enum 是枚举类型关键字，枚举类型名可以是任何合法的标识名，花括号内的枚举常量也可叫做枚举元素、枚举值。

例如：

```
enum month {January, February, March, April, May, June, July, August, Septemper, October,
        November, December};
```

上面语句定义了名为 month 的枚举类型，它的枚举常量是 January, February, March, April, May, June, July, August, Septemper, October, November, December。下面就可以引用上面的枚举类型来定义枚举变量了。

例如：

```
enum year this_month, next_month;
```

上面的语句定义了 2 个枚举变量 this_month、next_month，它们的值只能是 January 到 December。例如：

```
this_month=June; next_month=July;
```

在定义枚举类型的时候，也可以将定义枚举类型和定义枚举变量的语句写在一处，例如：

```
enum month {January, February, March, April, May, June, July, August, Septemper, October,
```

```
November, December} this_month, next_month;
```

对枚举类型的几点说明：

1）枚举常量表中的每一个元素都是定值（常量），不能随意进行赋值运算。枚举常量的值是在编译时得到的，默认情况下，C 语言系统将按照枚举类型定义时的顺序自动为每一个枚举元素赋初值 0、l、2、3、…。

例如：

```
enum month {January, February, March, April, May, June, July, August, Septemper, October,
        November, December};
```

该语句中，如若没有特别声明，各枚举元素的值为：

January=0，February=1，March=2，April=3，May=4，June=5，July=6，August=7，September=8，October=9，Novermber=10，December=11。

2）C 语言也允许在定义枚举类型时自行指定各枚举常量的值。除了被指定值的枚举常量外，其余枚举常量将依照其前一个值顺序递增。

例如：

```
enum month {January=13, February, March, April, May, June, July, August, Septemper, October,
        November, December};
```

该语句中有 January=13，由此依次递加，则有：February=14,…,December=24。

3）枚举类型变量不能被直接赋值。枚举类型数据比较特殊，如要赋值必须经过强制类型转换。如有下列枚举类型数据：

```
enum fruit{apple, pear, orange} fruit1, fruit2;
```

为枚举类型变量赋值的常见错误例子：

```
fruit1=2; fruit2=1;
```

为枚举类型变量正确赋值方法是：

```
fruit1=(enum fruit)2; fruit2=(enum fruit)1;
```

上面第一条语句是将枚举类型 fruit 顺序号为 2 的枚举常量 orange 赋给 fruitl，第二条语句是将顺序号为 1 的枚举常量 pear 赋给 fruit2。它们相当于：

```
fruit1=orange; fruit2=pear;
```

4）枚举类型数据可以进行关系比较运算，常用于程序中的判断语句。其比较方法是：默认情况下，按各元素定义的顺序号比较大小；如指定了某些枚举元素的值，比较方法

等同简单类型数据。

0.10.8 用 typedef 定义数据类型

为了适应不同用户的习惯，C 语言允许用户使用 typedef 将已有的类型标识符定义成新的类型标识符。

类型定义的一般形式为：

```
typedef  类型名  标识符;
```

其中，typedef 是类型定义关键字；“类型名”是已有的类型标识符，如 int、char、struct A 等，甚至可以是前面用 typedef 定义的类型标识；“标识符”是用户自定义的新标识符，用来当作新的类型名，常常用大写字母表示。

注意：用 typedef 类型定义以后．并未产生新的数据类型，只是允许用户使用新定义的“标识符”来表示原来的类型名。原来的类型标识符依然有效。

例如：

```
typedef int INTEGER;
typedef float FL;
typedef struct {float x,y;} ST;
```

有了上面的类型定义后，就可以用新的类型标识符建立变量了。例如：

```
INTEGER  i,j;          /*等价于 int i,j;*/
FL * p1;               /*等价于 float * p1;*/
ST a, b;               /*等价于 struct{float x,y;} a,b;
```

用 typdef 进行类型定义一般需要分三步完成：

1）按照常规方法写出定义变量的语句。例如：

```
float x,y;
```

2）对原有的类型名进行类型定义。先将变量名换成新类型名，然后在最前面加上 typedef。例如：

```
typedef     float  FL;
```

类型定义后，就可以用新的类型名 FL（一般用大写字母表示）替代原来的类型名 float 了。

3）用新类型名定义变量。例如：

```
FL x,y;
```

至此，类型定义完毕。

在进行 C 程序设计的时候，适当合理的使用 typedef 进行类型定义可以使程序的可移植性大大增强。不过，有几点必须注意：

1）用 typdef 进行类型定义，并不能产生新的类型，也不能取代原有的类型名，它只是为原有的类型名增加一个新的替换名而已。

2）用 typedef 只能定义各种类型名，不能用来定义变量。

3）C 语言允许用户用 typdef 嵌套定义新类型名，即：用 tydef 定义了一个新的类型名后，还可以用 typedef 将已定义的新类型名定义成另一个新类型名。

4）用 typedef 既可以定义简单数据类型，也可以定义复杂一些的构造数据类型，如数组、指针、结构、联合等。

5）类型定义语句 typedef 和预定义语句#define 在使用的时候有一些类似的地方，要注意区分，例如：

```
typedef float FL;
#define FL float;
```

二者的共同点都是用 FL 表示 float，不同之处在于前者是为原有的类型名增加一个新的替换名，只能用于类型定义，在编译时完成；后者仅仅是字符串的简单替换，其作用一目了然，在预编译时完成。

0.10.9 位运算

前面介绍的各种运算都是以字节作为最基本位进行的。但在很多系统程序中常要求在位(bit)一级进行运算或处理。C 语言提供了位运算的功能， 这使得 C 语言也能像汇编语言一样用来编写系统程序。

C 语言提供了六种位运算符，它们分别是：

& 按位与

| 按位或

^ 按位异或

~ 取反

<< 左移

>> 右移

1、按位与运算

按位与运算符"&"是双目运算符。其功能是参与运算的两数各对应的二进位相与。只有对应的两个二进位均为 1 时，结果位才为 1 ，否则为 0。参与运算的数以补码方式出现。

例如：9&5 可写算式如下： 00001001 (9 的二进制补码)&00000101 (5 的二进制补码) 00000001 (1 的二进制补码)，可见 9&5=1。

按位与运算通常用来对某些位清 0 或保留某些位。

2、按位或运算

按位或运算符“|”是双目运算符。其功能是参与运算的两数各对应的二进位相或。只要对应的二个二进位有一个为 1 时，结果位就为 1。参与运算的两个数均以补码出现。

例如：9|5 可写算式如下： 00001001|00000101=00001101，可见 9|5=13。

3、按位异或运算

按位异或运算符“^”是双目运算符。其功能是参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为 1。参与运算数仍以补码出现，例如 9^5 可写成算式如下： 00001001^00000101 = 00001100 ，可见 9^5=12。

4、求反运算 求反运算符～为单目运算符，具有右结合性。 其功能是对参与运算的数的各二进位按位求反。例如～9 的运算为： ~(00001001)结果为：11110110。

5、左移运算

左移运算符“<<”是双目运算符。其功能把“<< ”左边的运算数的各二进位全部左移若干位，由“<<”右边的数指定移动的位数，高位丢弃，低位补 0。例如： a<<4 指把 a 的各二进位向左移动 4 位。

如 a=00000011(十进制 3)，左移 4 位后为 00110000(十进制 48)。

6、右移运算

右移运算符“>>”是双目运算符。其功能是把“>> ”左边的运算数的各二进位全部右移若干位，“>>”右边的数指定移动的位数。

例如：设 a=15，a>>2 表示把 000001111 右移为 00000011(十进制 3)。 应该说明的是，对于有符号数，在右移时，符号位将随同移动。当为正数时， 最高位补 0，而为负数时，符号位为 1，最高位是补 0 或是补 1 取决于编译系统的规定。gcc 和很多系统规定为补 1。

例 0.36：位运算符的应用。

```
main()
{
        int a=9, b=5, c=-17;
        printf("a&b=%d, a&c=%d, b&c=%d \n",a&b,a&c,b&c);
        printf("a|b=%d, a|c=%d, b|c=%d \n",a|b,a|c,b|c);
        printf("a^b=%d, a^c=%d, b^c=%d\n",a^b,a^c,b^c);
        printf("~a=%d, ~b=%d, ~c=%d\n",~a,~b,~c);
        printf("a<<4=%d, b<<4=%d, c<<4=%d\n",a<<4,b<<4,c<<4);
        printf("a>>4=%d, b>>4=%d, c>>4=%d\n",a>>4,b>>4,c>>4);
}
```

上述程序输出了用不同的运算符对 a、b、c 进行运算的结果，程序运行结果如下：

```
a&b=1, a&c=9, b&c=5
a|b=13, a|c=-17, b|c=-17
a^b=12, a^c=-26, b^c=-22
~a=-10, ~b=-6, ~c=16
a<<4=144, b<<4=80, c<<4=-272
a>>4=0, b>>4=0, c>>4=-2
```